The Linux Command Line

Third Internet Edition

William Shotts

A LinuxCommand.org Book

Copyright ©2008-2016, William E. Shotts, Jr.

This work is licensed under the Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 United States License. To view a copy of this license, visit the link above or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042.

A version of this book is also available in printed form, published by No Starch Press. Copies may be purchased wherever fine books are sold. No Starch Press also offers electronic formats for popular e-readers. They can be reached at: https://www.nostarch.com.

Linux is the registered trademark of Linus Torvalds. All other trademarks belong to their respective owners.

This book is part of the LinuxCommand.org project, a site for Linux education and advocacy devoted to helping users of legacy operating systems migrate into the future. You may contact the LinuxCommand.org project at http://linuxcommand.org.

Version	Date	Description
16.07	July 28, 2016	Third Internet Edition.
13.07	July 6, 2013	Second Internet Edition.
09.12	December 14, 2009	First Internet Edition.

Why Use The Command Line?....xvi
What This Book Is About....xvii
Who Should Read This Book....xvii
What's In This Book....xviii
How To Read This Book....xviii
Prerequisites....xix
Why I Don't Call It “GNU/Linux”....xix
Acknowledgments....xx
First Internet Edition....xx
Second Internet Edition....xx
Third Internet Edition....xxi
Your Feedback Is Needed!....xxi
What's New In The Third Internet Edition....xxi
Further Reading....xxi
Colophon....xxi

Part 1 – Learning The Shell....1

Terminal Emulators....2
Your First Keystrokes....2
Command History....3
Cursor Movement....3
A Few Words About Mice And Focus....3
Try Some Simple Commands....4
Ending A Terminal Session....5
The Console Behind The Curtain....5
Summing Up....5
Further Reading....6

Understanding The File System Tree....7
The Current Working Directory....7
Listing The Contents Of A Directory....8
Changing The Current Working Directory....9
Absolute Pathnames....9

Relative Pathnames....9
Some Helpful Shortcuts....11
Important Facts About Filenames....11
Summing Up....12

! " # $

More Fun With ls....13
Options And Arguments....14
A Longer Look At Long Format....16
Determining A File's Type With file....17
Viewing File Contents With less....17
What Is “Text”?....17
Less Is More....19
A Guided Tour....19
Symbolic Links....23
Hard Links....24
Summing Up....24
Further Reading....24

% & " ' ( ) *

Wildcards....25
Character Ranges....27
Wildcards Work In The GUI Too....27
mkdir – Create Directories....28
cp – Copy Files And Directories....28
Useful Options And Examples....29
mv – Move And Rename Files....30
Useful Options And Examples....30
rm – Remove Files And Directories....31
Useful Options And Examples....32
Be Careful With rm!....32
ln – Create Links....33
Hard Links....33
Symbolic Links....34
Let's Build A Playground....34
Creating Directories....34
Copying Files....35
Moving And Renaming Files....36
Creating Hard Links....37
Creating Symbolic Links....38
Removing Files And Directories....39
Creating Symlinks With The GUI....41
Summing Up....41
Further Reading....41

* + $$ %

What Exactly Are Commands?....42
Identifying Commands....43
type – Display A Command's Type....43
which – Display An Executable's Location....43

Getting A Command's Documentation....44
help – Get Help For Shell Builtins....44
--help – Display Usage Information....45
man – Display A Program's Manual Page....45
apropos – Display Appropriate Commands....47
whatis – Display A Very Brief Description Of A Command....47
The Most Brutal Man Page Of Them All....48
info – Display A Program's Info Entry....48
README And Other Program Documentation Files....49
Creating Your Own Commands With alias....50
Summing Up....52
Further Reading....52

Standard Input, Output, And Error....53
Redirecting Standard Output....54
Redirecting Standard Error....55
Redirecting Standard Output And Standard Error To One File....56
Disposing Of Unwanted Output....57
/dev/null In Unix Culture....57
Redirecting Standard Input....57
cat – Concatenate Files....58
Pipelines....60
The Difference Between > and |....60
Filters....61
uniq - Report Or Omit Repeated Lines....61
wc – Print Line, Word, And Byte Counts....62
grep – Print Lines Matching A Pattern....62
head / tail – Print First / Last Part Of Files....63
tee – Read From Stdin And Output To Stdout And Files....64
Summing Up....65
Linux Is About Imagination....65

Expansion....67
Pathname Expansion....68
Pathname Expansion Of Hidden Files....69
Tilde Expansion....69
Arithmetic Expansion....70
Brace Expansion....71
Parameter Expansion....72
Command Substitution....73
Quoting....74
Double Quotes....75
Single Quotes....77
Escaping Characters....77
Backslash Escape Sequences....78
Summing Up....78
Further Reading....79

. (/ # + .0

Command Line Editing....80
Cursor Movement....80
Modifying Text....81
Cutting And Pasting (Killing And Yanking) Text....81
The Meta Key....82
Completion....82
Programmable Completion....84
Using History....84
Searching History....85
History Expansion....87
script....87
Summing Up....88
Further Reading....88

1 2 $ .1

Owners, Group Members, And Everybody Else....90
Reading, Writing, And Executing....91
chmod – Change File Mode....93
What The Heck Is Octal?....94
Setting File Mode With The GUI....96
umask – Set Default Permissions....97
Some Special Permissions....99
Changing Identities....100
su – Run A Shell With Substitute User And Group IDs....101
sudo – Execute A Command As Another User....102
Ubuntu And sudo....103
chown – Change File Owner And Group....103
chgrp – Change Group Ownership....104
Exercising Our Privileges....105
Changing Your Password....107
Summing Up....108
Further Reading....108

0 2 01

How A Process Works....109
Viewing Processes....110
Viewing Processes Dynamically With top....112
Controlling Processes....114
Interrupting A Process....115
Putting A Process In The Background....115
Returning A Process To The Foreground....116
Stopping (Pausing) A Process....117
Signals....118
Sending Signals To Processes With kill....118
Sending Signals To Multiple Processes With killall....121
Shutting Down The System....121
More Process Related Commands....122
Summing Up....123

Part 2 – Configuration And The Environment.....125

! $,

What Is Stored In The Environment?.....126
Examining The Environment.....126
Some Interesting Variables.....128
How Is The Environment Established?.....129
What's In A Startup File?.....130
Modifying The Environment.....132
Which Files Should We Modify?.....132
Text Editors.....132
Using A Text Editor.....133
Why Comments Are Important.....136
Activating Our Changes.....137
Summing Up.....137
Further Reading.....137

(3 .

Why We Should Learn vi.....138
A Little Background.....139
Starting And Stopping vi.....139
Compatibility Mode.....140
Editing Modes.....141
Entering Insert Mode.....142
Saving Our Work.....142
Moving The Cursor Around.....143
Basic Editing.....144
Appending Text.....144
Opening A Line.....145
Deleting Text.....146
Cutting, Copying, And Pasting Text.....147
Joining Lines.....149
Search-And-Replace.....149
Searching Within A Line.....149
Searching The Entire File.....149
Global Search-And-Replace.....150
Editing Multiple Files.....152
Switching Between Files.....153
Opening Additional Files For Editing.....153
Copying Content From One File Into Another.....154
Inserting An Entire File Into Another.....155
Saving Our Work.....156
Summing Up.....157
Further Reading.....157

$ 4 2 $" *.

Anatomy Of A Prompt.....158
Trying Some Alternative Prompt Designs.....160
Adding Color.....161

Terminal Confusion....162
Moving The Cursor....164
Saving The Prompt....165
Summing Up....166
Further Reading....166

Part 3 – Common Tasks And Essential Tools....167

% 2 + & $,.

Packaging Systems....168
How A Package System Works....169
Package Files....169
Repositories....169
Dependencies....170
High And Low-level Package Tools....170
Common Package Management Tasks....171
Finding A Package In A Repository....171
Installing A Package From A Repository....171
Installing A Package From A Package File....172
Removing A Package....172
Updating Packages From A Repository....173
Upgrading A Package From A Package File....173
Listing Installed Packages....174
Determining If A Package Is Installed....174
Displaying Info About An Installed Package....174
Finding Which Package Installed A File....175
Summing Up....175
The Linux Software Installation Myth....176
Further Reading....177

* & .

Mounting And Unmounting Storage Devices....178
Viewing A List Of Mounted File Systems....180
Why Unmounting Is Important....183
Determining Device Names....184
Creating New File Systems....187
Manipulating Partitions With fdisk....187
Creating A New File System With mkfs....190
Testing And Repairing File Systems....191
What The fsck?....192
Formatting Floppy Disks....192
Moving Data Directly To/From Devices....192
Creating CD-ROM Images....193
Creating An Image Copy Of A CD-ROM....193
Creating An Image From A Collection Of Files....194
A Program By Any Other Name....194
Writing CD-ROM Images....194
Mounting An ISO Image Directly....195
Blanking A Re-Writable CD-ROM....195

Writing An Image....195
Summing Up....196
Further Reading....196
Extra Credit....196

, 5 + 1.

Examining And Monitoring A Network....199
ping....199
traceroute....200
ip....201
netstat....202
Transporting Files Over A Network....203
ftp....203
lftp – A Better ftp....205
wget....205
Secure Communication With Remote Hosts....206
ssh....206
Tunneling With SSH....210
scp And sftp....210
An SSH Client For Windows?....212
Summing Up....212
Further Reading....212

' '

locate – Find Files The Easy Way....213
Where Does The locate Database Come From?....215
find – Find Files The Hard Way....215
Tests....216
Operators....218
Predefined Actions....221
User-Defined Actions....223
Improving Efficiency....224
xargs....224
Dealing With Funny Filenames....225
A Return To The Playground....225
Options....228
Summing Up....229
Further Reading....229

. ((6 +" 0

Compressing Files....230
gzip....231
bzip2....233
Don’t Be Compressive Compulsive....234
Archiving Files....234
tar....234
zip....240
Synchronizing Files And Directories....242
Using rsync Over A Network....244
Summing Up....245

Further Reading....246

1 - ! " %

What Are Regular Expressions?....247
grep....247
Metacharacters And Literals....249
The Any Character....250
Anchors....251
A Crossword Puzzle Helper....251
Bracket Expressions And Character Classes....252
Negation....252
Traditional Character Ranges....253
POSIX Character Classes....254
Reverting To Traditional Collation Order....257
POSIX Basic Vs. Extended Regular Expressions....258
POSIX....258
Alternation....259
Quantifiers....260
? - Match An Element Zero Or One Time....260
* - Match An Element Zero Or More Times....261
+ - Match An Element One Or More Times....262
{ } - Match An Element A Specific Number Of Times....262
Putting Regular Expressions To Work....263
Validating A Phone List With grep....263
Finding Ugly Filenames With find....264
Searching For Files With locate....265
Searching For Text With less And vim....265
Summing Up....267
Further Reading....267

0 2 ,.

Applications Of Text....268
Documents....269
Web Pages....269
Email....269
Printer Output....269
Program Source Code....269
Revisiting Some Old Friends....269
cat....270
MS-DOS Text Vs. Unix Text....271
sort....272
uniq....279
Slicing And Dicing....280
cut....281
Expanding Tabs....283
paste....284
join....285
Comparing Text....288
comm....288
diff....289

patch....291
Editing On The Fly....293
tr....293
ROT13: The Not-So-Secret Decoder Ring....294
sed....295
People Who Like sed Also Like....303
aspell....303
Summing Up....307
Further Reading....307
Extra Credit....308

' $ 7 " 01

Simple Formatting Tools....309
nl – Number Lines....309
fold – Wrap Each Line To A Specified Length....313
fmt – A Simple Text Formatter....313
pr – Format Text For Printing....317
printf – Format And Print Data....318
Document Formatting Systems....321
groff....322
Summing Up....328
Further Reading....328

2 0

A Brief History Of Printing....330
Printing In The Dim Times....330
Character-based Printers....331
Graphical Printers....332
Printing With Linux....333
Preparing Files For Printing....333
pr – Convert Text Files For Printing....333
Sending A Print Job To A Printer....335
lpr – Print Files (Berkeley Style)....335
lp – Print Files (System V Style)....336
Another Option: a2ps....337
Monitoring And Controlling Print Jobs....340
lpstat – Display Print System Status....340
lpq – Display Printer Queue Status....341
lprm / cancel – Cancel Print Jobs....342
Summing Up....342
Further Reading....342

$" 2 $ %

What Is Compiling?....343
Are All Programs Compiled?....344
Compiling A C Program....345
Obtaining The Source Code....345
Examining The Source Tree....347
Building The Program....349
Installing The Program....353

Summing Up....353
Further Reading....353

Part 4 – Writing Shell Scripts....355

% 8 ' " *,

What Are Shell Scripts?....356
How To Write A Shell Script....356
Script File Format....357
Executable Permissions....358
Script File Location....358
Good Locations For Scripts....360
More Formatting Tricks....360
Long Option Names....360
Indentation And line-continuation....360
Configuring vim For Script Writing....361
Summing Up....362
Further Reading....362

* (2 9 ,

First Stage: Minimal Document....363
Second Stage: Adding A Little Data....365
Variables And Constants....366
Assigning Values To Variables And Constants....369
Here Documents....370
Summing Up....373
Further Reading....373

, ":) 5) %

Shell Functions....375
Local Variables....378
Keep Scripts Running....379
Shell Functions In Your .bashrc File....382
Summing Up....382
Further Reading....382

' 5 ; 6 .

if....383
Exit Status....384
test....386
File Expressions....386
String Expressions....389
Integer Expressions....390
A More Modern Version Of test....391
(( )) - Designed For Integers....393
Combining Expressions....394
Portability Is The Hobgoblin Of Little Minds....396
Control Operators: Another Way To Branch....397

Summing Up....398
Further Reading....398

. - / # " %00

read – Read Values From Standard Input....401
Options....403
IFS....405
You Can’t Pipe read....407
Validating Input....407
Menus....409
Summing Up....410
Extra Credit....411
Further Reading....411

1 ' 5 ; < " 5 = %

Looping....412
while....412
Breaking Out Of A Loop....415
until....416
Reading Files With Loops....417
Summing Up....418
Further Reading....418

0 % 1

Syntactic Errors....419
Missing Quotes....420
Missing Or Unexpected Tokens....421
Unanticipated Expansions....421
Logical Errors....423
Defensive Programming....423
Verifying Input....425
Design Is A Function Of Time....425
Testing....425
Test Cases....426
Debugging....427
Finding The Problem Area....427
Tracing....428
Examining Values During Execution....430
Summing Up....430
Further Reading....431

' 5 ; 6 %

case....432
Patterns....434
Performing Multiple Actions....436
Summing Up....437
Further Reading....437

2 2 $ % 1

Accessing The Command Line....439
Determining The Number of Arguments....440
shift – Getting Access To Many Arguments....441
Simple Applications....442
Using Positional Parameters With Shell Functions....443
Handling Positional Parameters En Masse....444
A More Complete Application....446
Summing Up....449
Further Reading....452

' 5 ; < " %*

for: Traditional Shell Form....453
Why i?....455
for: C Language Form....456
Summing Up....457
Further Reading....458

% ($ %*1

Parameter Expansion....459
Basic Parameters....459
Expansions To Manage Empty Variables....460
Expansions That Return Variable Names....462
String Operations....462
Case Conversion....465
Arithmetic Evaluation And Expansion....467
Number Bases....468
Unary Operators....468
Simple Arithmetic....468
Assignment....470
Bit Operations....472
Logic....473
bc – An Arbitrary Precision Calculator Language....476
Using bc....477
An Example Script....478
Summing Up....479
Extra Credit....479
Further Reading....479

* (# %.

What Are Arrays?....481
Creating An Array....481
Assigning Values To An Array....482
Accessing Array Elements....483
Array Operations....485
Outputting The Entire Contents Of An Array....485
Determining The Number Of Array Elements....485
Finding The Subscripts Used By An Array....486
Adding Elements To The End Of An Array....486
Sorting An Array....487
Deleting An Array....487

Associative Arrays....488
Summing Up....489
Further Reading....489

, ! %10

Group Commands And Subshells....490
Process Substitution....494
Traps....496
Temporary Files....498
Asynchronous Execution....499
wait....499
Named Pipes....501
Setting Up A Named Pipe....501
Using Named Pipes....502
Summing Up....502
Further Reading....502

*0%

I want to tell you a story.

No, not the story of how, in 1991, Linus Torvalds wrote the first version of the Linux kernel. You can read that story in lots of Linux books. Nor am I going to tell you the story of how, some years earlier, Richard Stallman began the GNU Project to create a free Unix-like operating system. That's an important story too, but most other Linux books have that one, as well.

No, I want to tell you the story of how you can take back control of your computer.

When I began working with computers as a college student in the late 1970s, there was a revolution going on. The invention of the microprocessor had made it possible for ordinary people like you and me to actually own a computer. It's hard for many people today to imagine what the world was like when only big business and big government ran all the computers. Let's just say, you couldn't get much done.

Today, the world is very different. Computers are everywhere, from tiny wristwatches to giant data centers to everything in between. In addition to ubiquitous computers, we also have a ubiquitous network connecting them together. This has created a wondrous new age of personal empowerment and creative freedom, but over the last couple of decades something else has been happening. A few giant corporations have been imposing their control over most of the world's computers and deciding what you can and cannot do with them. Fortunately, people from all over the world are doing something about it. They are fighting to maintain control of their computers by writing their own software. They are building Linux.

Many people speak of “freedom” with regard to Linux, but I don't think most people know what this freedom really means. Freedom is the power to decide what your computer does, and the only way to have this freedom is to know what your computer is doing. Freedom is a computer that is without secrets, one where everything can be known if you care enough to find out.

Why Use The Command Line?

Have you ever noticed in the movies when the “super hacker,”—you know, the guy who can break into the ultra-secure military computer in under thirty seconds—sits down at the computer, he never touches a mouse? It's because movie makers realize that we, as human beings, instinctively know the only way to really get anything done on a computer

is by typing on a keyboard!

Most computer users today are only familiar with the (GUI) and have been taught by vendors and pundits that the (CLI) is a terrifying thing of the past. This is unfortunate, because a good command line interface is a marvelously expressive way of communicating with a computer in much the same way the written word is for human beings. It's been said that “graphical user interfaces make easy tasks easy, while command line interfaces make difficult tasks possible” and this is still very true today.

Since Linux is modeled after the Unix family of operating systems, it shares the same rich heritage of command line tools as Unix. Unix came into prominence during the early 1980s (although it was first developed a decade earlier), before the widespread adoption of the graphical user interface and, as a result, developed an extensive command line interface instead. In fact, one of the strongest reasons early adopters of Linux chose it over, say, Windows NT was the powerful command line interface which made the “difficult tasks possible.”

What This Book Is About

This book is a broad overview of “living” on the Linux command line. Unlike some books that concentrate on just a single program, such as the shell program, , this book will try to convey how to get along with the command line interface in a larger sense. How does it all work? What can it do? What's the best way to use it?

While any serious discussion of the command line will invariably lead to system administration topics, this book only touches on a few administration issues. It will, however, prepare the reader for additional study by providing a solid foundation in the use of the command line, an essential tool for any serious system administration task.

Many other books try to broaden their appeal by including other platforms such as generic Unix and OS X. In doing so, they “water down” their content to feature only general topics. This book, on the other hand, only covers contemporary Linux distributions. Ninety-five percent of the content is useful for users of other Unix-like systems, but this book is highly targeted at the modern Linux command line user.

Who Should Read This Book

This book is for new Linux users who have migrated from other platforms. Most likely you are a “power user” of some version of Microsoft Windows. Perhaps your boss has told you to administer a Linux server, or maybe you're just a desktop user who is tired of all the security problems and want to give Linux a try. That's fine. All are welcome here.

That being said, there is no shortcut to Linux enlightenment. Learning the command line is challenging and takes real effort. It's not that it's so hard, but rather it's so . The av-

erage Linux system has literally of programs you can employ on the command line. Consider yourself warned; learning the command line is not a casual endeavor.

On the other hand, learning the Linux command line is extremely rewarding. If you think you're a “power user” now, just wait. You don't know what real power is—yet. And, unlike many other computer skills, knowledge of the command line is long lasting. The skills learned today will still be useful ten years from now. The command line has survived the test of time.

It is also assumed that you have no programming experience, but not to worry, we'll start you down that path as well.

What's In This Book

This material is presented in a carefully chosen sequence, much like a tutor sitting next to you guiding you along. Many authors treat this material in a “systematic” fashion, which makes sense from a writer’s perspective, but can be very confusing to new users.

Another goal is to acquaint you with the Unix way of thinking, which is different from the Windows way of thinking. Along the way, we'll go on a few side trips to help you understand why certain things work the way they do and how they got that way. Linux is not just a piece of software, it's also a small part of the larger Unix culture, which has its own language and history. I might throw in a rant or two, as well.

This book is divided into four parts, each covering some aspect of the command line experience:

starts our exploration of the basic language of the command line including such things as the structure of commands, file system navigation, command line editing, and finding help and documentation for commands.

! " # $ covers editing configuration files that control the computer's operation from the command line.

% ! # $ explores many of the ordinary tasks that are commonly performed from the command line. Unix-like operating systems, such as Linux, contain many “classic” command line programs that are used to perform powerful operations on data.

& ' (introduces shell programming, an admittedly rudimentary, but easy to learn, technique for automating many common computing tasks. By learning shell programming, you will become familiar with concepts that can be applied to many other programming languages.

How To Read This Book

Start at the beginning of the book and follow it to the end. It isn’t written as a reference work, it's really more like a story with a beginning, middle, and an end.

Prerequisites

To use this book, all you will need is a working Linux installation. You can get this in one of two ways:

1.) * +, (It doesn't matter which distribution you choose, though most people today start out with either Ubuntu, Fedora, or OpenSUSE. If in doubt, try Ubuntu first. Installing a modern Linux distribution can be ridiculously easy or ridiculously difficult depending on your hardware. I suggest a desktop computer that is a couple of years old and has at least 256 megabytes of RAM and 6 gigabytes of free hard disk space. Avoid laptops and wireless networks if at all possible, as these are often more difficult to get working.
2. - . !/0 - 1 " One of the cool things you can do with many Linux distributions is run them directly from a CDROM (or USB flash drive) without installing them at all. Just go into your BIOS setup and set your computer to "Boot from CDROM," insert the live CD, and reboot. Using a live CD is a great way to test a computer for Linux compatibility prior to installation. The disadvantage of using a live CD is that it may be very slow compared to having Linux installed on your hard drive. Both Ubuntu and Fedora (among others) have live CD versions.

Regardless of how you install Linux, you will need to have occasional superuser (i.e., administrative) privileges to carry out the lessons in this book.

After you have a working installation, start reading and follow along with your own computer. Most of the material in this book is "hands on," so sit down and get typing!

Why I Don't Call It "GNU/Linux"

In some quarters, it's politically correct to call the Linux operating system the "GNU/Linux operating system." The problem with "Linux" is that there is no completely correct way to name it because it was written by many different people in a vast, distributed development effort. Technically speaking, Linux is the name of the operating system's kernel, nothing more. The kernel is very important of course, since it makes the operating system go, but it's not enough to form a complete operating system.

Enter Richard Stallman, the genius-philosopher who founded the Free Software movement, started the Free Software Foundation, formed the GNU Project, wrote the first version of the GNU C Compiler (gcc), created the GNU General Public License (the GPL), etc., etc., etc. He that you call it "GNU/Linux" to properly reflect the contributions of the GNU Project. While the GNU Project predates the Linux kernel, and the project's contributions are extremely deserving of recog-

nition, placing them in the name is unfair to everyone else who made significant contributions. Besides, I think “Linux/GNU” would be more technically accurate since the kernel boots first and everything else runs on top of it.

In popular usage, “Linux” refers to the kernel and all the other free and open source software found in the typical Linux distribution; that is, the entire Linux ecosystem, not just the GNU components. The operating system marketplace seems to prefer one-word names such as DOS, Windows, macOS, Solaris, Irix, AIX. I have chosen to use the popular format. If, however, you prefer to use “GNU/Linux” instead, please perform a mental search-and-replace while reading this book. I won't mind.

Acknowledgments

I want to thank the following people, who helped make this book possible:

First Internet Edition

Jenny Watson, Acquisitions Editor at Wiley Publishing who originally suggested that I write a shell scripting book.

John C. Dvorak, noted columnist and pundit. In an episode of his video podcast, “Cranky Geeks,” Mr. Dvorak described the process of writing: “Hell. Write 200 words a day and in a year, you have a novel.” This advice led me to write a page a day until I had a book.

Dmitri Popov wrote an article in Free Software Magazine titled, “Creating a book template with Writer,” which inspired me to use OpenOffice.org Writer for composing the text. As it turned out, it worked wonderfully.

Mark Polesky performed an extraordinary review and test of the text.

Jesse Becker, Tomasz Chrzczonowicz, Michael Levin, Spence Miner also tested and reviewed portions of the text.

Karen M. Shotts contributed a lot of hours, polishing my so-called English by editing the text.

Second Internet Edition

Special thanks go out to the following individuals who provided valuable feedback incorporated into the Second Internet Edition: Adrian Arpidez, Hu Bo, Heriberto Cantú, Joshua Escamilla, Bruce Fowler, Ma Jun, Seth King, Mike O'Donnell, Parviz Rasoulipour, Gabriel Stutzman, and Christian Wuethrich.

Third Internet Edition

Special thanks go out to the following individuals who provided valuable feedback incorporated into the Third Internet Edition: Steve Bragg, Lixin Duan, Sunil Joshi, Chris Knight, Jim Kovacs, Bartłomiej Majka, Bashar Maree, Frank McTipps, Justin Page, Waldo Ribeiro, Satej Kumar Sahu, Mikhail Sizov, Pickles Spill, Francesco Turco, Wolfram Volpi, and Boyang Wang .

And lastly, many thanks to the readers of LinuxCommand.org, who have sent me so many kind emails. Their encouragement gave me the idea that I was really on to something!

Your Feedback Is Needed!

This book is an ongoing project, like many open source software projects. If you find a technical error, drop me a line at:

bshotts@users.sourceforge.net

Your changes and suggestions may get into future releases.

What's New In The Third Internet Edition

This version of has undergone some additional refinement, clarification, and modernization. I also fixed a few bugs ;-).

Further Reading

Here are some Wikipedia articles about the famous people mentioned above:
http://en.wikipedia.org/wiki/Linus_Torvalds
http://en.wikipedia.org/wiki/Richard_Stallman

The Free Software Foundation and the GNU Project:
http://en.wikipedia.org/wiki/Free_Software_Foundation
http://www.fsf.org
http://www.gnu.org

Richard Stallman has written extensively on the “GNU/Linux” naming issue:
http://www.gnu.org/gnu/why-gnu-linux.html
http://www.gnu.org/gnu/gnu-linux-faq.html#tools

Colophon

This book was originally written using OpenOffice.org Writer in Liberation Serif and Sans fonts on a Dell Inspiron 530N, factory configured with Ubuntu 8.04. The PDF version of the text was generated directly by OpenOffice.org Writer. The Second Internet Edition was produced on the same computer using LibreOffice Writer on Ubuntu 12.04. The Third Internet Edition was produced with LibreOffice Writer on a System76 Ratel

Pro computer, factory configured with Ubuntu 14.04.

Part 1 – Learning The Shell

When we speak of the command line, we are really referring to the . The shell is a program that takes keyboard commands and passes them to the operating system to carry out. Almost all Linux distributions supply a shell program from the GNU Project called . The name “bash” is an acronym for “Bourne Again SHell”, a reference to the fact is an enhanced replacement for , the original Unix shell program written by Steve Bourne.

Terminal Emulators

When using a graphical user interface, we need another program called a to interact with the shell. If we look through our desktop menus, we will probably find one. KDE uses and GNOME uses , though it's likely called simply “terminal” on our menu. There are a number of other terminal emulators available for Linux, but they all basically do the same thing; give us access to the shell. You will probably develop a preference for one or another based on the number of bells and whistles it has.

Your First Keystrokes

So let's get started. Launch the terminal emulator! Once it comes up, we should see something like this:

This is called a and it will appear whenever the shell is ready to accept input. While it may vary in appearance somewhat depending on the distribution, it will usually include your , followed by the current working directory (more about that in a little bit) and a dollar sign.

If the last character of the prompt is a pound sign (“#”) rather than a dollar sign, the terminal session has privileges. This means either we are logged in as the root user or we selected a terminal emulator that provides superuser (administrative) privi-

leges.

Assuming that things are good so far, let's try some typing. Enter some gibberish at the prompt like so:

```

```

Since this command makes no sense, the shell will tell us so and give us another chance:

```

```

Command History

If we press the up-arrow key, we will see that the previous command “kaekfjaeifj” reappears after the prompt. This is called . Most Linux distributions remember the last 1000 commands by default. Press the down-arrow key and the previous command disappears.

Cursor Movement

Recall the previous command with the up-arrow key again. Now try the left and right-arrow keys. See how we can position the cursor anywhere on the command line? This makes editing commands easy.

A Few Words About Mice And Focus

While the shell is all about the keyboard, you can also use a mouse with your terminal emulator. There is a mechanism built into the X Window System (the underlying engine that makes the GUI go) that supports a quick copy and paste technique. If you highlight some text by holding down the left mouse button and dragging the mouse over it (or double clicking on a word), it is copied into a buffer maintained by X. Pressing the middle mouse button will cause the text to be pasted at the cursor location. Try it.

2 : Don't be tempted to use and to perform copy and paste inside a terminal window. They don't work. These control codes have different meanings to the shell and were assigned many years before Microsoft Windows.

Your graphical desktop environment (most likely KDE or GNOME), in an effort to behave like Windows, probably has its set to “click to focus.” This means for a window to get focus (become active) you need to click on it. This is contrary to the traditional X behavior of “focus follows mouse” which means that a window gets focus just by passing the mouse over it. The window will not come to the foreground until you click on it but it will be able to receive input. Setting the focus policy to “focus follows mouse” will make the copy and paste technique even more useful. Give it a try if you can (some desktop environments such as Ubuntu's Unity no longer support it). I think if you give it a chance you will prefer it. You will find this setting in the configuration program for your window manager.

Try Some Simple Commands

Now that we have learned to type, let's try a few simple commands. The first one is . This command displays the current time and date.

```
      ! "# $% &" '() *$"+
```

A related command is which, by default, displays a calendar of the current month.

```
       , *$"+
( - ) . )   (
          "  *
 / & # + 0 ! %
"$ "" "* "/ "& "# "+
"0 "! "% *$ *" ** */
*& *# *+ *0 *! *% /$
/"
```

To see the current amount of free space on your disk drives, enter :

```
   ,          "1         2   3       2  4 -
5  5  *         "#""#&#*   #$"*/%*   %%&%0"+  /&4 5
5  5  #          #%+/"%$!  *+#&#&*&  /$$$!&/*  &04 5
```

```
5  5  "          "&00+&    "0/0$   "**0+#  "/4 5
 6              *#+!#+       $   *#+!#+   $4 5  5
```

Likewise, to display the amount of free memory, enter the command.

```
-     #"/0"*    #$/%0+      %0/+        $      #/"*    "**%"+
 57       5      /0#0&!    "/0%+&
(8 6   "$#**&!    "$&0"*    %&0#/+
```

Ending A Terminal Session

We can end a terminal session by either closing the terminal emulator window, or by entering the command at the shell prompt:

```
```

The Console Behind The Curtain

Even if we have no terminal emulator running, several terminal sessions continue to run behind the graphical desktop. Called or , these sessions can be accessed on most Linux distributions by pressing 3 " through 3 + . When a session is accessed, it presents a login prompt into which we can enter our username and password. To switch from one virtual console to another, press 3 and " - + . To return to the graphical desktop, press 3 0 .

Summing Up

As we begin our journey, we are introduced to the shell and see the command line for the first time and learn how to start and end a terminal session. We also see how to issue some simple commands and perform a little light command line editing. That wasn't so scary was it?

Further Reading

To learn more about Steve Bourne, father of the Bourne Shell, see this Wikipedia article:
http://en.wikipedia.org/wiki/Steve_Bourne

Here is an article about the concept of shells in computing:
http://en.wikipedia.org/wiki/Shell_(computing)

The first thing we need to learn (besides just typing) is how to navigate the file system on our Linux system. In this chapter we will introduce the following commands:

68 - Print name of current working directory

- Change directory

- List directory contents

Understanding The File System Tree

Like Windows, a Unix-like operating system such as Linux organizes its files in what is called a . This means that they are organized in a tree-like pattern of directories (sometimes called folders in other systems), which may contain files and other directories. The first directory in the file system is called the . The root directory contains files and subdirectories, which contain more files and subdirectories and so on and so on.

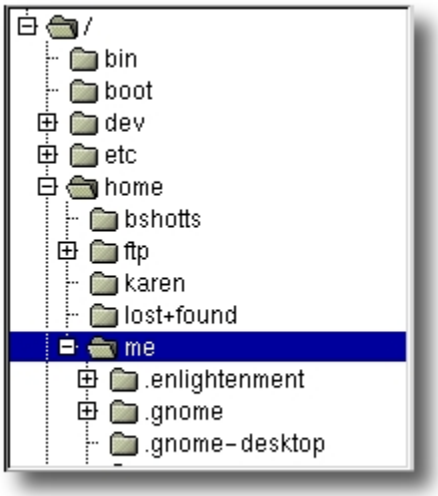
Note that unlike Windows, which has a separate file system tree for each storage device, Unix-like systems such as Linux always have a single file system tree, regardless of how many drives or storage devices are attached to the computer. Storage devices are attached (or more correctly,) at various points on the tree according to the whims of the , the person (or persons) responsible for the maintenance of the system.

The Current Working Directory

Most of us are probably familiar with a graphical file manager which represents the file system tree as in Figure 1. Notice that the tree is usually shown upended, that is, with the root at the top and the various branches descending below.

However, the command line has no pictures, so to navigate the file system tree we need to think of it in a different way.

Imagine that the file system is a maze shaped like an upside-down tree and we are able to



!

stand in the middle of it. At any given time, we are inside a single directory and we can see the files contained in the directory and the pathway to the directory above us (called the) and any subdirectories below us. The directory we are standing in is called the . To display the current working directory, we use the 68 (print working directory) command.

```
5  5
```

When we first log in to our system (or start a terminal emulator session) our current working directory is set to our . Each user account is given its own home directory and it is the only place a regular user is allowed to write files.

Listing The Contents Of A Directory

To list the files and directories in the current working directory, we use the command.

```
9  6 9    -  :    :  ) 6   ;
```

Actually, we can use the command to list the contents of any directory, not just the current working directory, and there are many other fun things it can do as well. We'll spend more time with in the next chapter.

Changing The Current Working Directory

To change your working directory (where we are standing in our tree-shaped maze) we use the command. To do this, type followed by the of the desired working directory. A pathname is the route we take along the branches of the tree to get to the directory we want. Pathnames can be specified in one of two different ways; as ! or as . Let's deal with absolute pathnames first.

Absolute Pathnames

An absolute pathname begins with the root directory and follows the tree branch by branch until the path to the desired directory or file is completed. For example, there is a directory on your system in which most of our system's programs are installed. The pathname of the directory is 5 5 . This means from the root directory (represented by the leading slash in the pathname) there is a directory called "usr" which contains a directory called "bin".

```
5 5


<<<=        ,>   ,     <<<
```

Now we can see that we have changed the current working directory to 5 5 and that it is full of files. Notice how the shell prompt has changed? As a convenience, it is usually set up to automatically display the name of the working directory.

Relative Pathnames

Where an absolute pathname starts from the root directory and leads to its destination, a relative pathname starts from the working directory. To do this, it uses a couple of special notations to represent relative positions in the file system tree. These special notations are "." (dot) and ".." (dot dot).

The "." notation refers to the working directory and the ".." notation refers to the working directory's parent directory. Here is how it works. Let's change the working directory to

5 5 again:

```
5 5
```

Okay, now let's say that we wanted to change the working directory to the parent of 5 5 which is 5 . We could do that two different ways. Either with an absolute pathname:

```
5
```

Or, with a relative pathname:

```
5
```

Two different methods with identical results. Which one should we use? The one that requires the least typing!

Likewise, we can change the working directory from 5 to 5 5 in two different ways. Either using an absolute pathname:

```
5 5
```

Or, with a relative pathname:

```
5 5
```

Now, there is something important that I must point out here. In almost all cases, we can

omit the "./". It is implied. Typing:

does the same thing. In general, if we do not specify a pathname to something, the working directory will be assumed.

Some Helpful Shortcuts

In Table 2-1 we see some useful ways the current working directory can be quickly changed.

! " #

Shortcut	Result
	Changes the working directory to your home directory.
	Changes the working directory to the previous working directory.
	Changes the working directory to the home directory of $. For example, will change the directory to the home directory of user “bob.”

Important Facts About Filenames

1. Filenames that begin with a period character are hidden. This only means that will not list them unless you say . When your account was created, several hidden files were placed in your home directory to configure things for your account. Later on we will take a closer look at some of these files to see how you can customize your environment. In addition, some applications place their configuration and settings files in your home directory as hidden files.
2. Filenames and commands in Linux, like Unix, are case sensitive. The filenames “File1” and “file1” refer to different files.
3. Linux has no concept of a “file extension” like some other operating systems. You may name files any way you like. The contents and/or purpose of a file is

determined by other means. Although Unix-like operating systems don't use file extensions to determine the contents/purpose of files, many application programs do.

4. Though Linux supports long filenames which may contain embedded spaces and punctuation characters, limit the punctuation characters in the names of files you create to period, dash, and underscore. % & ! ' If you want to represent spaces between words in a filename, use underscore characters. You will thank yourself later.

Summing Up

In this chapter we saw how the shell treats the directory structure of the system. We learned about absolute and relative pathnames and the basic commands that are used to move about that structure. In the next chapter we will use this knowledge to go on a tour of a modern Linux system.

! " # $

Now that we know how to move around the file system, it's time for a guided tour of our Linux system. Before we start however, we're going to learn some more commands that will be useful along the way:

– List directory contents

– Determine file type

– View file contents

More Fun With

The command is probably the most used command, and for good reason. With it, we can see directory contents and determine a variety of important file and directory attributes. As we have seen, we can simply enter to see a list of files and subdirectories contained in the current working directory:

```
9  6 9   -  :   :  ) 6   ;
```

Besides the current working directory, we can specify the directory to list, like so:

```
                    6
```

Or even specify multiple directories. In this example we will list both the user's home directory (symbolized by the "~" character) and the 5 directory:

```
5  5
```

```
9   6 9     -   :     :    ) 6    ;


5

                      6
```

We can also change the format of the output to reveal more detail:

```
   #+
 8 8   *    &$%+ *$$0 "$ *+ "0 *$ 9    6
 8 8   *    &$%+ *$$0 "$ *+ "0 *$ 9
 8 8   *    &$%+ *$$0 "$ *+ "0 *$ -
 8 8   *    &$%+ *$$0 "$ *+ "0 *$ :
 8 8   *    &$%+ *$$0 "$ *+ "0 *$ :
 8 8   *    &$%+ *$$0 "$ *+ "0 *$ ) 6
 8 8   *    &$%+ *$$0 "$ *+ "0 *$ ;
```

By adding “-l” to the command, we changed the output to the long format.

Options And Arguments

This brings us to a very important point about how most commands work. Commands are often followed by one or more that modify their behavior, and further, by one or more , the items upon which the command acts. So most commands look kind of like this:

```

```

Most commands use options consisting of a single character preceded by a dash, for example, “-l”, but many commands, including those from the GNU Project, also support , consisting of a word preceded by two dashes. Also, many commands allow multiple short options to be strung together. In this example, the command is given two options, the “l” option to produce long format output, and the “t” option to sort the result by the file's modification time.

```

```

We'll add the long option “--reverse” to reverse the order of the sort:

Note that command options, like filenames in Linux, are case-sensitive.

The command has a large number of possible options. The most common are listed in Table 3-1.

! ( )

Option	Long Option	Description
		List all files, even those with names that begin with a period, which are normally not listed (i.e., hidden).
3		Like the option above except it does not list < (current directory) and << (parent directory).
	,	Ordinarily, if a directory is specified, will list the contents of the directory, not the directory itself. Use this option in conjunction with the option to see details about the directory rather than its contents.
	,	This option will append an indicator character to the end of each listed name. For example, a “/” if the name is a directory.
		In long format listings, display file sizes in human readable format rather than in bytes.
		Display results in long format.
		Display the results in reverse order. Normally, displays its results in ascending alphabetical order.
(		Sort results by file size.
		Sort by modification time.

A Longer Look At Long Format

As we saw before, the “-l” option causes to display its results in long format. This format contains a great deal of useful information. Here is the ' 6 directory from an early Ubuntu system:

```
8      "        /#0+*%+ *$$0 $& $/ "" $# ' 6          <
8      "        ""!+*"% *$$0 $& $/ "" $#            <6
8      "         &0#!& *$$0 $& $/ "" $#     '     <6
8      "         &&/## *$$0 $& $/ "" $#     1     <6
8      "         /&/%" *$$0 $& $/ "" $#     2    <6
8      "         /*$#% *$$0 $& $/ "" $#          <
8      "        "#%0&& *$$0 $& $/ "" $#           <
8      "         *0!/0 *$$0 $& $/ "" $#      8  <
8      "         %!!"+ *$$0 $& $/ "" $#       <
8      "        &#/0+& *$$0 $& $/ "" $#    8     <
8      "        /#!/0& *$$0 $& $/ "" $#      (  <
```

Let's look at the different fields from one of the files and examine their meanings:

! ("

Field	Meaning
8	Access rights to the file. The first character indicates the type of file. Among the different types, a leading dash means a regular file, while a “d” indicates a directory. The next three characters are the access rights for the file's owner, the next three are for members of the file's group, and the final three are for everyone else. The full meaning of this is discussed in Chapter 9 – Permissions.
"	File's number of hard links. See the discussion of links later in this chapter.
	The username of the file's owner.
	The name of the group which owns the file.
/*$#%	Size of the file in bytes.
*$$0 $& $/ "" $#	Date and time of the file's last modification.
<	Name of the file.

Determining A File's Type With

As we explore the system it will be useful to know what files contain. To do this we will use the command to determine a file's type. As we discussed earlier, filenames in Linux are not required to reflect a file's contents. While a filename like “picture.jpg” would normally be expected to contain a JPEG compressed image, it is not required to in Linux. We can invoke the command this way:

```

```

When invoked, the command will print a brief description of the file's contents. For example:

```
6    < 6   :'?       >  @       "<$"
```

There are many kinds of files. In fact, one of the common ideas in Unix-like operating systems such as Linux is that “everything is a file.” As we proceed with our lessons, we will see just how true that statement is.

While many of the files on your system are familiar, for example MP3 and JPEG, there are many kinds that are a little less obvious and a few that are quite strange.

Viewing File Contents With

The command is a program to view text files. Throughout our Linux system, there are many files that contain human-readable text. The program provides a convenient way to examine them.

What Is “Text”?

There are many ways to represent information on a computer. All methods involve defining a relationship between the information and some numbers that will be used to represent it. Computers, after all, only understand numbers and all data is converted to numeric representation.

Some of these representation systems are very complex (such as compressed video files), while others are rather simple. One of the earliest and simplest is

called *# ++ . ASCII (pronounced "As-Key") is short for American Standard Code for Information Interchange. This is a simple encoding scheme that was first used on Teletype machines to map keyboard characters to numbers.

Text is a simple one-to-one mapping of characters to numbers. It is very compact. Fifty characters of text translates to fifty bytes of data. It is important to understand that text only contains a simple mapping of characters to numbers. It is not the same as a word processor document such as one created by Microsoft Word or LibreOffice Writer. Those files, in contrast to simple ASCII text, contain many non-text elements that are used to describe its structure and formatting. Plain ASCII text files contain only the characters themselves and a few rudimentary control codes like tabs, carriage returns and line feeds.

Throughout a Linux system, many files are stored in text format and there are many Linux tools that work with text files. Even Windows recognizes the importance of this format. The well-known NOTEPAD.EXE program is an editor for plain ASCII text files.

Why would we want to examine text files? Because many of the files that contain system settings (called) are stored in this format, and being able to read them gives us insight about how the system works. In addition, some of the actual programs that the system uses (called) are stored in this format. In later chapters, we will learn how to edit text files in order to modify systems settings and write our own scripts, but for now we will just look at their contents.

The command is used like this:

Once started, the program allows us to scroll forward and backward through a text file. For example, to examine the file that defines all the system's user accounts, enter the following command:

Once the program starts, we can view the contents of the file. If the file is longer than one page, we can scroll up and down. To exit , press the “q” key.

The table below lists the most common keyboard commands used by .

! ((

Command	Action
Page Up or	Scroll back one page
Page Down or space	Scroll forward one page
Up Arrow	Scroll up one line
Down Arrow	Scroll down one line
?	Move to the end of the text file
"? or	Move to the beginning of the text file
5	Search forward to the next occurrence of
	Search for the next occurrence of the previous search
	Display help screen
A	Quit

Less Is More

The program was designed as an improved replacement of an earlier Unix program called . The name “less” is a play on the phrase “less is more”—a motto of modernist architects and designers.

falls into the class of programs called “pagers,” programs that allow the easy viewing of long text documents in a page by page manner. Whereas the program could only page forward, the program allows paging both forward and backward and has many other features as well.

A Guided Tour

The file system layout on a Linux system is much like that found on other Unix-like systems. The design is actually specified in a published standard called the , # . Not all Linux distributions conform to the standard exactly but most come pretty close.

Next, we are going to wander around the file system ourselves to see what makes our

Linux system tick. This will give us a chance to practice our navigation skills. One of the things we will discover is that many of the interesting files are in plain human-readable text. As we go about our tour, try the following:

1. into a given directory
2. List the directory contents with
3. If you see an interesting file, determine its contents with
4. If it looks like it might be text, try viewing it with

((3 If you are using a mouse, you can double click on a filename to copy it and middle click to paste it into commands.

As we wander around, don't be afraid to look at stuff. Regular users are largely prohibited from messing things up. That's the system administrator's job! If a command complains about something, just move on to something else. Spend some time looking around. The system is ours to explore. Remember, in Linux, there are no secrets!

Table 3-4 lists just a few of the directories we can explore. There may be some slight differences depending on the distribution. Don't be afraid to look around and try more!

! (- .) #

Directory	Comments
5	The root directory. Where everything begins.
5	Contains binaries (programs) that must be present for the system to boot and run.
5	Contains the Linux kernel, initial RAM disk image (for drivers needed at boot time), and the boot loader. Interesting files: 5 5 5 < or < , which are used to configure the boot loader. 5 5 B (or something similar), the Linux kernel
5	This is a special directory which contains . “Everything is a file” also applies to devices. Here is where the kernel maintains a list of all the devices it understands.

Directory	Comments
5	The 5 directory contains all of the system-wide configuration files. It also contains a collection of shell scripts which start each of the system services at boot time. Everything in this directory should be readable text. Interesting files: While everything in 5 is interesting, here are some all-time favorites: 5 5 , a file that defines when automated jobs will run. 5 5 , a table of storage devices and their associated mount points. 5 56 8 , a list of the user accounts.
5	In normal configurations, each user is given a directory in 5 . Ordinary users can only write files in their home directories. This limitation protects the system from errant user activity.
5	Contains shared library files used by the core system programs. These are similar to DLLs in Windows.
5 7	Each formatted partition or device using a Linux file system, such as ext3, will have this directory. It is used in the case of a partial recovery from a file system corruption event. Unless something really bad has happened to your system, this directory will remain empty.
5	On modern Linux systems the 5 directory will contain the mount points for removable media such as USB drives, CD-ROMs, etc. that are mounted automatically at insertion.
5	On older Linux systems, the 5 directory contains mount points for removable devices that have been mounted manually.
5 6	The 5 6 directory is used to install “optional” software. This is mainly used to hold commercial software products that may be installed on your system.

Directory	Comments
56	The 56 directory is special. It's not a real file system in the sense of files stored on your hard drive. Rather, it is a virtual file system maintained by the Linux kernel. The "files" it contains are peepholes into the kernel itself. The files are readable and will give you a picture of how the kernel sees your computer.
5	This is the home directory for the root account.
5	This directory contains "system" binaries. These are programs that perform vital system tasks that are generally reserved for the superuser.
5 6	The 5 6 directory is intended for storage of temporary, transient files created by various programs. Some configurations cause this directory to be emptied each time the system is rebooted.
5	The 5 directory tree is likely the largest one on a Linux system. It contains all the programs and support files used by regular users.
5 5	5 5 contains the executable programs installed by your Linux distribution. It is not uncommon for this directory to hold thousands of programs.
5 5	The shared libraries for the programs in 5 5 .
5 5	The 5 5 tree is where programs that are not included with your distribution but are intended for system-wide use are installed. Programs compiled from source code are normally installed in 5 5 5 . On a newly installed Linux system, this tree exists, but it will be empty until the system administrator puts something in it.
5 5	Contains more system administration programs.
5 5	5 5 contains all the shared data used by programs in 5 5 . This includes things like default configuration files, icons, screen backgrounds, sound files, etc.
5 5 5	Most packages installed on the system will include some kind of documentation. In 5 5 5 , we will find documentation files organized by package.

Directory	Comments
5	With the exception of 5 6 and 5 , the directories we have looked at so far remain relatively static, that is, their contents don't change. The 5 directory tree is where data that is likely to change is stored. Various databases, spool files, user mail, etc. are located here.
5 5	5 5 contains , records of various system activity. These are very important and should be monitored from time to time. The most useful ones are 5 5 5 and/or 5 5 5 , . Note that for security reasons on some systems, you must be the superuser to view log files.

Symbolic Links

As we look around, we are likely to see a directory listing with an entry like this:

```
8 8 8 "        "" *$$0 $! "" $0 /&   < <+ C    *<+<
```

Notice how the first letter of the listing is “l” and the entry seems to have two filenames? This is a special kind of a file called a ! (also known as a or). In most Unix-like systems it is possible to have a file referenced by multiple names. While the value of this may not be obvious, it is really a useful feature.

Picture this scenario: A program requires the use of a shared resource of some kind contained in a file named “foo,” but “foo” has frequent version changes. It would be good to include the version number in the filename so the administrator or other interested party could see what version of “foo” is installed. This presents a problem. If we change the name of the shared resource, we have to track down every program that might use it and change it to look for a new resource name every time a new version of the resource is installed. That doesn't sound like fun at all.

Here is where symbolic links save the day. Let's say we install version 2.6 of “foo,” which has the filename “foo-2.6” and then create a symbolic link simply called “foo” that points to “foo-2.6.” This means that when a program opens the file “foo”, it is actually opening the file “foo-2.6”. Now everybody is happy. The programs that rely on “foo” can find it and we can still see what actual version is installed. When it is time to upgrade to “foo-2.7,” we just add the file to our system, delete the symbolic link “foo” and create a new one that points to the new version. Not only does this solve the problem of the version upgrade, but it also allows us to keep both versions on our machine. Imagine that “foo-2.7” has a bug (damn those developers!) and we need to revert to the old version.

Again, we just delete the symbolic link pointing to the new version and create a new symbolic link pointing to the old version.

The directory listing above (from the 5 directory of a Fedora system) shows a symbolic link called “libc.so.6” that points to a shared library file called “libc-2.6.so.” This means that programs looking for “libc.so.6” will actually get the file “libc-2.6.so.” We will learn how to create symbolic links in the next chapter.

Hard Links

While we are on the subject of links, we need to mention that there is a second type of link called a . Hard links also allow files to have multiple names, but they do it in a different way. We’ll talk more about the differences between symbolic and hard links in the next chapter.

Summing Up

With our tour behind us, we have learned a lot about our system. We've seen various files and directories and their contents. One thing you should take away from this is how open the system is. In Linux there are many important files that are plain human-readable text. Unlike many proprietary systems, Linux makes everything available for examination and study.

Further Reading

The full version of the , # can be found here:
http://www.pathname.com/fhs/

An article about the directory structure of Unix and Unix-like systems:
http://en.wikipedia.org/wiki/Unix_directory_structure

A detailed description of the ASCII text format:
http://en.wikipedia.org/wiki/ASCII

% & " ' ( )

At this point, we are ready for some real work! This chapter will introduce the following commands:

6 – Copy files and directories

– Move/rename files and directories

– Create directories

– Remove files and directories

– Create hard and symbolic links

These five commands are among the most frequently used Linux commands. They are used for manipulating both files and directories.

Now, to be frank, some of the tasks performed by these commands are more easily done with a graphical file manager. With a file manager, we can drag and drop a file from one directory to another, cut and paste files, delete files, etc. So why use these old command line programs?

The answer is power and flexibility. While it is easy to perform simple file manipulations with a graphical file manager, complicated tasks can be easier with the command line programs. For example, how could we copy all the HTML files from one directory to another, but only copy files that do not exist in the destination directory or are newer than the versions in the destination directory? Pretty hard with a file manager. Pretty easy with the command line:

Wildcards

Before we begin using our commands, we need to talk about a shell feature that makes these commands so powerful. Since the shell uses filenames so much, it provides special characters to help you rapidly specify groups of filenames. These special characters are

called . Using wildcards (which is also known as !!) allows you to select filenames based on patterns of characters. The table below lists the wildcards and what they select:

! - /

Wildcard	Meaning
D	Matches any characters
E	Matches any single character
	Matches any character that is a member of the set
F	Matches any character that is not a member of the set
	Matches any character that is a member of the specified

Table 4-2 lists the most commonly used character classes:

! - " 0

Character Class	Meaning
	Matches any alphanumeric character
6	Matches any alphabetic character
	Matches any numeral
8	Matches any lowercase letter
66	Matches any uppercase letter

Using wildcards makes it possible to construct very sophisticated selection criteria for filenames. Here are some examples of patterns and what they match:

! - (/ 1

Pattern	Matches
D	All files
D	Any file beginning with “g”
D<	Any file beginning with “b” followed by any characters and ending with “.txt”

9 EEE	Any file beginning with “Data” followed by exactly three characters
D	Any file beginning with either an “a”, a “b”, or a “c”
G3 12:< $ % $ % $ %	Any file beginning with “BACKUP.” followed by exactly three numerals
66 D	Any file beginning with an uppercase letter
F D	Any file not beginning with a numeral
D 8 "*/	Any file ending with a lowercase letter or the numerals “1”, “2”, or “3”

Wildcards can be used with any command that accepts filenames as arguments, but we’ll talk more about that in Chapter 7.

Character Ranges

If you are coming from another Unix-like environment or have been reading some other books on this subject, you may have encountered the 3 H or the B character range notations. These are traditional Unix notations and worked in older versions of Linux as well. They can still work, but you have to be very careful with them because they will not produce the expected results unless properly configured. For now, you should avoid using them and use character classes instead.

Wildcards Work In The GUI Too

Wildcards are especially valuable not only because they are used so frequently on the command line, but are also supported by some graphical file managers.

In 2 (the file manager for GNOME), you can select files using the Edit/Select Pattern menu item. Just enter a file selection pattern with wildcards and the files in the currently viewed directory will be highlighted for selection.

In some versions of / (and 4 5 (the file managers for KDE), you can enter wildcards directly on the location bar. For example, if you want to see all the files starting with a lowercase “u” in the /usr/bin directory, enter “/usr/bin/u*” in the location bar and it will display the result.

Many ideas originally found in the command line interface make their way into the graphical interface, too. It is one of the many things that make the Linux desktop so powerful.

– Create Directories

The command is used to create directories. It works like this:

```

```

6 When three periods follow an argument in the description of a command (as above), it means that the argument can be repeated, thus:

```

```

would create a single directory named “dir1”, while

```

```

would create three directories named “dir1”, “dir2”, and “dir3”.

– Copy Files And Directories

The 6 command copies files or directories. It can be used two different ways:

```

```

to copy the single file or directory “item1” to file or directory “item2” and:

```

```

to copy multiple items (either files or directories) into a directory.

Useful Options And Examples

Here are some of the commonly used options (the short option and the equivalent long option) for 6 :

! - -)

Option	Meaning
>	Copy the files and directories and all of their attributes, including ownerships and permissions. Normally, copies take on the default attributes of the user performing the copy.
>	Before overwriting an existing file, prompt the user for confirmation.)" ((" 7 + + "
>	Recursively copy directories and their contents. This option (or the option) is required when copying directories.
> 6	When copying files from one directory to another, only copy files that either don't exist, or are newer than the existing corresponding files, in the destination directory. This is useful when copying large numbers of file as it skips over files that don't need to be copied.
>	Display informative messages as the copy is performed.

! - 2 1

Command	Results
6	Copy to " .)" 7 + + " If " does not exist, it is created.
6	Same as above, except that if " exists, the user is prompted before it is overwritten.
6	Copy and " into directory . must already exist.
6 "5D *	Using a wildcard, all the files in are copied into " . " must already exist.

6	Copy the contents of directory to directory " . If directory " does not exist, it is created and, after the copy, will contain the same contents as directory . If directory " does exist, then directory (and its contents) will be copied into " .

– Move And Rename Files

The command performs both file moving and file renaming, depending on how it is used. In either case, the original filename no longer exists after the operation. is used in much the same way as 6 :

```
```

to move or rename file or directory “item1” to “item2” or:

```
```

to move one or more items from one directory to another.

Useful Options And Examples

shares many of the same options as 6 :

! - 3)

Option	Meaning
>	Before overwriting an existing file, prompt the user for confirmation.)" ((" 7 + + "
> 6	When moving files from one directory to another, only move files that either don't exist, or are newer than the existing corresponding files in the destination directory.
>	Display informative messages as the move is performed.

! - 4 1

Command	Results
	Move to " .)" 7 + + " If " does not exist, it is created.) 7
	Same as above, except that if " exists, the user is prompted before it is overwritten.
	Move and " into directory . must already exist.
	If directory " does not exist, create directory " and move the contents of directory into " and delete directory . If directory " does exist, move directory (and its contents) into directory " .

– Remove Files And Directories

The command is used to remove (delete) files and directories:

```
```

where “item” is one or more files or directories.

Useful Options And Examples

Here are some of the common options for :

! - 5)

Option	Meaning
>	Before deleting an existing file, prompt the user for confirmation.)" ((" 7 + "
>	Recursively delete directories. This means that if a directory being deleted has subdirectories, delete them too. To delete a directory, this option must be specified.
>	Ignore nonexistent files and do not prompt. This overrides the option.
>	Display informative messages as the deletion is performed.

! - 6 1

Command	Results
	Delete silently.
	Same as above, except that the user is prompted for confirmation before the deletion is performed.
	Delete and and its contents.
	Same as above, except that if either or do not exist, will continue silently.

Be Careful With !

Unix-like operating systems such as Linux do not have an undelete command. Once you delete something with , it's gone. Linux assumes you're smart and you know what you're doing.

Be particularly careful with wildcards. Consider this classic example. Let's say you want to delete just the HTML files in a directory. To do this, you type:

D<

which is correct, but if you accidentally place a space between the “*” and the “.html” like so:

D <

the command will delete all the files in the directory and then complain that there is no file called “.html”.

" (Whenever you use wildcards with (besides carefully checking your typing!), test the wildcard first with . This will let you see the files that will be deleted. Then press the up arrow key to recall the command and replace the with .

– Create Links

The command is used to create either hard or symbolic links. It is used in one of two ways:

to create a hard link, and:

to create a symbolic link where “item” is either a file or a directory.

Hard Links

Hard links are the original Unix way of creating links, compared to symbolic links, which are more modern. By default, every file has a single hard link that gives the file its name. When we create a hard link, we create an additional directory entry for a file. Hard links have two important limitations:

1. A hard link cannot reference a file outside its own file system. This means a link cannot reference a file that is not on the same disk partition as the link itself.

2. A hard link may not reference a directory.

A hard link is indistinguishable from the file itself. Unlike a symbolic link, when you list a directory containing a hard link you will see no special indication of the link. When a hard link is deleted, the link is removed but the contents of the file itself continue to exist (that is, its space is not deallocated) until all links to the file are deleted.

It is important to be aware of hard links because you might encounter them from time to time, but modern practice prefers symbolic links, which we will cover next.

Symbolic Links

Symbolic links were created to overcome the limitations of hard links. Symbolic links work by creating a special type of file that contains a text pointer to the referenced file or directory. In this regard, they operate in much the same way as a Windows shortcut though of course, they predate the Windows feature by many years ;-)

A file pointed to by a symbolic link, and the symbolic link itself are largely indistinguishable from one another. For example, if you write something to the symbolic link, the referenced file is written to. However when you delete a symbolic link, only the link is deleted, not the file itself. If the file is deleted before the symbolic link, the link will continue to exist, but will point to nothing. In this case, the link is said to be ! . In many implementations, the command will display broken links in a distinguishing color, such as red, to reveal their presence.

The concept of links can seem very confusing, but hang in there. We're going to try all this stuff and it will, hopefully, become clear.

Let's Build A Playground

Since we are going to do some real file manipulation, let's build a safe place to “play” with our file manipulation commands. First we need a directory to work in. We'll create one in our home directory and call it “playground.”

Creating Directories

The command is used to create a directory. To create our playground directory we will first make sure we are in our home directory and will then create the new directory:

```
!
```

To make our playground a little more interesting, let's create a couple of directories inside

it called “dir1” and “dir2”. To do this, we will change our current working directory to 6 , and execute another :

```
                    !
      6 ,
```

Notice that the command will accept multiple arguments allowing us to create both directories with a single command.

Copying Files

Next, let's get some data into our playground. We'll do this by copying a file. Using the 6 command, we'll copy the 6 8 file from the 5 directory to the current working directory:

```
      6 ,
```

Notice how we used the shorthand for the current working directory, the single trailing period. So now if we perform an , we will see our file:

```
      6 ,
   "*
 8 8   *     &$%+ *$"+ $" "$ "+ &$   "
 8 8   *     &$%+ *$"+ $" "$ "+ &$   *
 8     "     "+#$ *$"+ $" "$ "+ $0 6  8
```

Now, just for fun, let's repeat the copy using the “-v” option (verbose) to see what it does:

```
      6 ,
I5  56  8 J  C I<56  8 J
```

The 6 command performed the copy again, but this time displayed a concise message indicating what operation it was performing. Notice that 6 overwrote the first copy without any warning. Again this is a case of 6 assuming that you know what you’re are doing. To get a warning, we'll include the “-i” (interactive) option:

```
        6 ,
6    8   I<56  8 JE
```

Responding to the prompt by entering a “y” will cause the file to be overwritten, any other character (for example, “n”) will cause 6 to leave the file alone.

Moving And Renaming Files

Now, the name “passwd” doesn't seem very playful and this is a playground, so let's change it to something else:

```
        6 ,
```

Let's pass the fun around a little by moving our renamed file to each of the directories and back again:

```
        6 ,
```

to move it first to directory " , then:

```
        6 ,
```

to move it from " to * , then:

```
        6 ,
```

to finally bring it back to the current working directory. Next, let's see the effect of on directories. First we will move our data file into " again:

```
        6 ,
```

then move " into * and confirm it with :

```
      6 ,
      6 ,
  &
8 8   *       &$%+ *$$"+ $" "" $+ $+    "
      6 ,
  &
8     "       "+#$ *$"+ $" "$ "+ //
```

Note that since * already existed, moved " into * . If * had not existed, would have renamed " to * . Lastly, let's put everything back:

```
      6 ,
      6 ,
```

Creating Hard Links

Now we'll try some links. First the hard links. We'll create some links to our data file like so:

```
      6 ,
      6 ,
      6 ,
```

So now we have four instances of the file "fun". Let's take a look at our playground directory:

```
      6 ,
  "+
8 8   *       &$%+ *$"+ $" "& "+ "0   "
8 8   *       &$%+ *$"+ $" "& "+ "0   *
8     &       "+#$ *$"+ $" "$ "+ //
8     &       "+#$ *$"+ $" "$ "+ //
```

One thing you notice is that the second field in the listing for and both contain a "4" which is the number of hard links that now exist for the file. You'll remember that a file will aways have at least one link because the file's name is created by a link. So, how do we know that and are, in fact, the same file? In this case, is not very helpful. While we can see that and are both the same size (field 5), our listing provides no way to be sure. To solve this problem, we're

going to have to dig a little deeper.

When thinking about hard links, it is helpful to imagine that files are made up of two parts: the data part containing the file's contents and the name part which holds the file's name. When we create hard links, we are actually creating additional name parts that all refer to the same data part. The system assigns a chain of disk blocks to what is called an , which is then associated with the name part. Each hard link therefore refers to a specific inode containing the file's contents.

The command has a way to reveal this information. It is invoked with the “-i” option:

```
      6 ,
   "+
"*/#/#/%  8 8   *       &$%+ *$"+ $" "& "+ "0   "
"*/#/#&$  8 8   *       &$%+ *$"+ $" "& "+ "0   *
"*/#/#/!  8     &       "+#$ *$"+ $" "$ "+ //
"*/#/#/!  8     &       "+#$ *$"+ $" "$ "+ //
```

In this version of the listing, the first field is the inode number and, as we can see, both and share the same inode number, which confirms they are the same file.

Creating Symbolic Links

Symbolic links were created to overcome the two disadvantages of hard links: Hard links cannot span physical devices and hard links cannot reference directories, only files. Symbolic links are a special type of file that contains a text pointer to the target file or directory.

Creating symbolic links is similar to creating hard links:

```
      6 ,                              !
      6 ,                                  !
      6 ,                                  !
```

The first example is pretty straightforward, we simply add the “-s” option to create a symbolic link rather than a hard link. But what about the next two? Remember, when we create a symbolic link, we are creating a text description of where the target file is relative to the symbolic link. It's easier to see if we look at the output:

```
      6 ,
```

```
   &
8     &       "+#$ *$"+ $" "$ "+ //
8 8 8 "         + *$"+ $" "# "# "0     ,   C <<5
```

The listing for , in " shows that it is a symbolic link by the leading “l” in the first field and that it points to “../fun”, which is correct. Relative to the location of , , is in the directory above it. Notice too, that the length of the symbolic link file is 6, the number of characters in the string “../fun” rather than the length of the file to which it is pointing.

When creating symbolic links, you can either use absolute pathnames:

```
      6 ,                                  !              !
```

or relative pathnames, as we did in our earlier example. In most cases, using relative pathnames is more desirable because it allows a directory tree containing symbolic links and their referenced files to be renamed and/or moved without breaking the links.

In addition to regular files, symbolic links can also reference directories:

```
      6 ,                                  !
      6 ,
  "+
8 8   *       &$%+ *$"+ $" "# "# "0   "
8 8 8 "         & *$"+ $" "+ "& &#   " ,  C   "
8 8   *       &$%+ *$"+ $" "# "# "0   *
8     &       "+#$ *$"+ $" "$ "+ //
8     &       "+#$ *$"+ $" "$ "+ //
8 8 8 "         / *$"+ $" "# "# "#     ,  C
```

Removing Files And Directories

As we covered earlier, the command is used to delete files and directories. We are going to use it to clean up our playground a little bit. First, let's delete one of our hard links:

```
      6 ,
      6 ,
  "*
8 8   *       &$%+ *$"+ $" "# "# "0   "
8 8 8 "         & *$"+ $" "+ "& &#   " ,  C   "
```

```
8 8   *       &$%+ *$"+ $" "# "# "0   *
8     /       "+#$ *$"+ $" "$ "+ //
8 8 8 "         / *$"+ $" "# "# "#     ,  C
```

That worked as expected. The file is gone and the link count shown for is reduced from four to three, as indicated in the second field of the directory listing. Next, we'll delete the file , and just for enjoyment, we'll include the “-i” option to show what that does:

```
      6 ,
            I  JE
```

Enter “y” at the prompt and the file is deleted. But let's look at the output of now. Noticed what happened to , ? Since it's a symbolic link pointing to a now-nonexistent file, the link is ! :

```
      6 ,
  !
8 8   *       &$%+ *$"+ $" "# "# "0   "
8 8 8 "         & *$"+ $" "+ "& &#    " ,  C    "
8 8   *       &$%+ *$"+ $" "# "# "0   *
8 8 8 "         / *$"+ $" "# "# "#                          ,        C
```

Most Linux distributions configure to display broken links. On a Fedora box, broken links are displayed in blinking red text! The presence of a broken link is not in and of itself dangerous, but it is rather messy. If we try to use a broken link we will see this:

```
      6 ,                          !
  ,  K              ,
```

Let's clean up a little. We'll delete the symbolic links:

```
      6 ,                          !     !
      6 ,
  !
8 8   *       &$%+ *$"+ $" "# "# "0   "
8 8   *       &$%+ *$"+ $" "# "# "0   *
```

One thing to remember about symbolic links is that most file operations are carried out on the link's target, not the link itself. is an exception. When you delete a link, it is the link that is deleted, not the target.

Finally, we will remove our playground. To do this, we will return to our home directory and use rm with the recursive option (-r) to delete playground and all of its contents, including its subdirectories:

```
      6 ,
                    !
```

Creating Symlinks With The GUI

The file managers in both GNOME and KDE provide an easy and automatic method of creating symbolic links. With GNOME, holding the Ctrl+Shift keys while dragging a file will create a link rather than copying (or moving) the file. In KDE, a small menu appears whenever a file is dropped, offering a choice of copying, moving, or linking the file.

Summing Up

We've covered a lot of ground here and it will take a while to fully sink in. Perform the playground exercise over and over until it makes sense. It is important to get a good understanding of basic file manipulation commands and wildcards. Feel free to expand on the playground exercise by adding more files and directories, using wildcards to specify files for various operations. The concept of links is a little confusing at first, but take the time to learn how they work. They can be a real lifesaver.

Further Reading

A discussion of symbolic links: http://en.wikipedia.org/wiki/Symbolic_link

* + $$

Up to this point, we have seen a series of mysterious commands, each with its own mysterious options and arguments. In this chapter, we will attempt to remove some of that mystery and even create some of our own commands. The commands introduced in this chapter are:

,6 – Indicate how a command name is interpreted

8 – Display which executable program will be executed

6 – Get help for shell builtins

– Display a command's manual page

6 6 – Display a list of appropriate commands

– Display a command's info entry

8 – Display a very brief description of a command

– Create an alias for a command

What Exactly Are Commands?

A command can be one of four different things:

1. # (like all those files we saw in 5 5 . Within this category, programs can be ! such as programs written in C and C++, or programs written in such as the shell, perl, python, ruby, etc.
2. # " . supports a number of commands internally called ! . The command, for example, is a shell builtin.
3. # " These are miniature shell scripts incorporated into the . We will cover configuring the environment and writing shell functions in later chapters, but for now, just be aware that they exist.
4. # Commands that we can define ourselves, built from other commands.

Identifying Commands

It is often useful to know exactly which of the four kinds of commands is being used and Linux provides a couple of ways to find out.

,6 – Display A Command's Type

The ,6 command is a shell builtin that displays the kind of command the shell will execute, given a particular command name. It works like this:

```
!
```

where “command” is the name of the command you want to examine. Here are some examples:

```
                    !  !
,6
                    !
        I      L ,J
                    !
6  5  56
```

Here we see the results for three different commands. Notice that the one for (taken from a Fedora system) and how the command is actually an alias for the command with the “-- color=tty” option added. Now we know why the output from is displayed in color!

8 – Display An Executable's Location

Sometimes there is more than one version of an executable program installed on a system. While this is not very common on desktop systems, it's not unusual on large servers. To determine the exact location of a given executable, the 8 command is used:

```
5  5
```

8 only works for executable programs, not builtins nor aliases that are substitutes for actual executable programs. When we try to use 8 on a shell builtin, for example, , we either get no response or an error message:

```
5   5   58            M5 6 5   "<+<$N$/5    5   5   5A
/</5    5   5       5    5 6 5   "<+<$N$/5    5   5   5       5   5
   5    5   5    5    5    5  5   O
```

which is a fancy way of saying “command not found.”

Getting A Command's Documentation

With this knowledge of what a command is, we can now search for the documentation available for each kind of command.

6 – Get Help For Shell Builtins

has a built-in help facility available for each of the shell builtins. To use it, type “help” followed by the name of the shell builtin. For example:

```
       =P  :
            8            ,<

                    ,    9@Q<  )         9@Q
   RS-'            <

)          9:3)R                6                ,
        9@Q<  3                ,         9:3)R      6
 ,       M O< 3            ,
      ,<  @  9@Q       8          M5O>      9:3)R          <

@            ,          >            6    I     N    J
  >    8                            <  @
        >                 9@Q<

S6
 =          ,                  8
 :           6 ,           ,        8          8    ,

            :  6        66    >            8            ,
                              ,>     8        B

)                 8  ,          >      I =J 8    6      <

'   (
Q      $            ,        >       :.9
        , 8    :      T    B       8   <
```

6 When square brackets appear in the description of a command's syntax, they indicate optional items. A vertical bar character indicates mutually exclusive items. In the case of the command above:

=P :

This notation says that the command may be followed optionally by either a “-L” or a “-P” and further, if the “-P” option is specified the “-e” option may also be included followed by the optional argument “dir”.

While the output of 6 for the commands is concise and accurate, it is by no means tutorial and as we can see, it also seems to mention a lot of things we haven't talked about yet! Don't worry. We'll get there.

6 – Display Usage Information

Many executable programs support a “--help” option that displays a description of the command's supported syntax and options. For example:

```
2          S:)@SK  9@Q' )SQU<<<
        9@Q' )SQUM   O>      ,          ,     <

  H>        L SK)'V) M('=    O          ,          SK)'V)
-     ,                6              ,          6
  <
   >      L-S9'              M         O>     L 8  W
  6>   6                        >     6

   >            6                              ,
        6         6  ,       6
                  6


Q 6          X                          <   C<
```

Some programs don't support the “--help” option, but try it anyway. Often it results in an error message that will reveal the same usage information.

– Display A Program's Manual Page

Most executable programs intended for command line use provide a formal piece of documentation called a or . A special paging program called is used to view them. It is used like this:

where “program” is the name of the command to view.

Man pages vary somewhat in format but generally contain a title, a synopsis of the command's syntax, a description of the command's purpose, and a listing and description of each of the command's options. Man pages, however, do not usually include examples, and are intended as a reference, not a tutorial. As an example, let's try viewing the man page for the command:

On most Linux systems, uses to display the manual page, so all of the familiar commands work while displaying the page.

The “manual” that displays is broken into sections and not only covers user commands but also system administration commands, programming interfaces, file formats and more. The table below describes the layout of the manual:

! 2 % 7) 8

Section	Contents
1	User commands
2	Programming interfaces for kernel system calls
3	Programming interfaces to the C library
4	Special files such as device nodes and drivers
5	File formats
6	Games and amusements such as screen savers
7	Miscellaneous
8	System administration commands

Sometimes we need to look in a specific section of the manual to find what we are looking for. This is particularly true if we are looking for a file format that is also the name of a command. Without specifying a section number, we will always get the first instance of a match, probably in section 1. To specify a section number, we use like this:

For example:

```
"
```

This will display the man page describing the file format of the 5 56 8 file.

6 6 – Display Appropriate Commands

It is also possible to search the list of man pages for possible matches based on a search term. It's very crude but sometimes helpful. Here is an example of a search for man pages using the search term “floppy”:

```
                                        !
      N   66,N       M!O                        6
                          66,                -S(  ,6
                 M!O    = 8                66,
   66,              M!O           66,
    66,             M"O       6     66,                ?KS-'
                 M"O              66,    >
                                  3)
                 M"O          -(9S(      ,          8
                                66,
```

The first field in each line of output is the name of the man page, the second field shows the section. Note that the command with the “-k” option performs the exact same function as 6 6 .

8 – Display A Very Brief Description Of A Command

The 8 program displays the name and a one line description of a man page matching a specified keyword:

```
                 M"O               ,
```

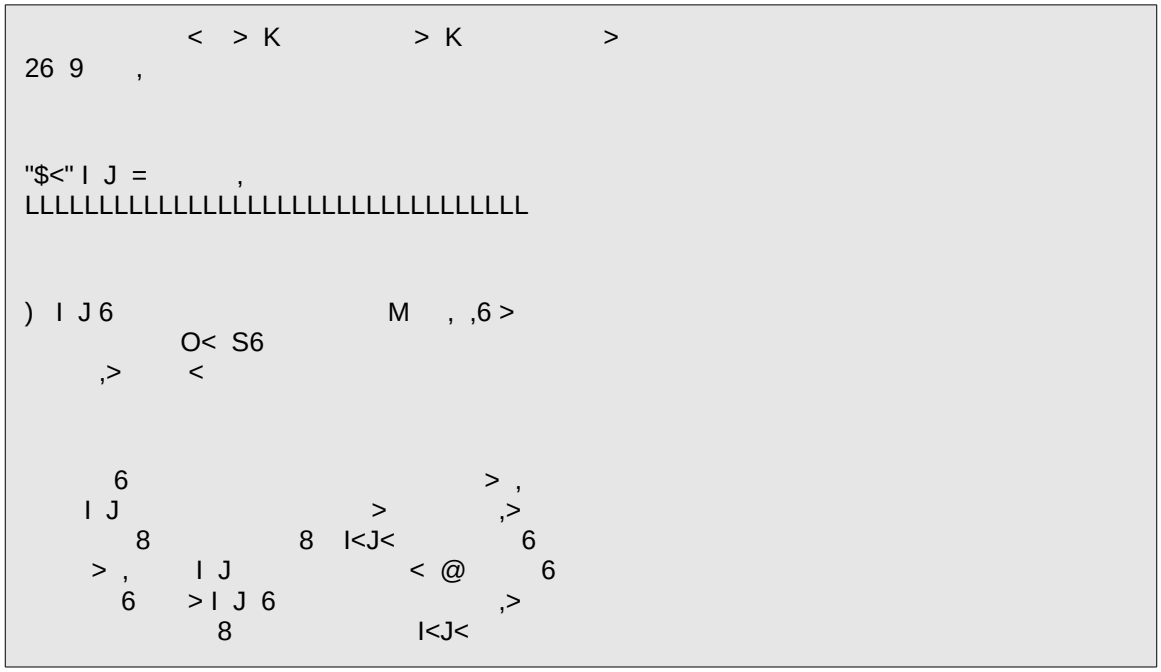
The Most Brutal Man Page Of Them All

As we have seen, the manual pages supplied with Linux and other Unix-like systems are intended as reference documentation and not as tutorials. Many man pages are hard to read, but I think that the grand prize for difficulty has got to go to the man page for . As I was doing my research for this book, I gave it careful review to ensure that I was covering most of its topics. When printed, it's over 80 pages long and extremely dense, and its structure makes absolutely no sense to a new user.

On the other hand, it is very accurate and concise, as well as being extremely complete. So check it out if you dare and look forward to the day when you can read it and it all makes sense.

– Display A Program's Info Entry

The GNU Project provides an alternative to man pages for their programs, called “info.” Info pages are displayed with a reader program named, appropriately enough, . Info pages are much like web pages. Here is a sample:



```
 G,      >      6          6       ,>
BB @    M       <   < BO          > +/       ) 6
```

The program reads , which are tree structured into individual , each containing a single topic. Info files contain hyperlinks that can move you from node to node. A hyperlink can be identified by its leading asterisk, and is activated by placing the cursor upon it and pressing the enter key.

To invoke , type “info” followed optionally by the name of a program. Below is a table of commands used to control the reader while displaying an info page:

! 2 "

Command	Action
E	Display command help
: 26 or G 6	Display previous page
: 9 or (6	Display next page
	Next - Display the next node
6	Previous - Display the previous node
	Up - Display the parent node of the currently displayed node, usually a menu.
'	Follow the hyperlink at the cursor location
A	Quit

Most of the command line programs we have discussed so far are part of the GNU Project's “coreutils” package, so typing:

```

```

will display a menu page with hyperlinks to each program contained in the coreutils package.

README And Other Program Documentation Files

Many software packages installed on your system have documentation files residing in the 5 5 5 directory. Most of these are stored in plain text format and can

be viewed with . Some of the files are in HTML format and can be viewed with a web browser. We may encounter some files ending with a “.gz” extension. This indicates that they have been compressed with the B 6 compression program. The gzip package includes a special version of called B that will display the contents of gzip-compressed text files.

Creating Your Own Commands With

Now for our very first experience with programming! We will create a command of our own using the command. But before we start, we need to reveal a small command line trick. It's possible to put more than one command on a line by separating each command with a semicolon character. It works like this:

```
        T          T          <<<
```

Here's the example we will use:

```
                  # #
            +&          6

5  5
```

As we can see, we have combined three commands on one line. First we change directory to 5 then list the directory and finally return to the original directory (by using J J) so we end up where we started. Now let's turn this sequence into a new command using . The first thing we have to do is dream up a name for our new command. Let's try “test”. Before we do that, it would be a good idea to find out if the name “test” is already being used. To find out, we can use the ,6 command again:

```
                  !
```

Oops! The name “test” is already taken. Let's try “foo”:

```
                  !
   ,6
```

Great! “foo” is not taken. So let's create our alias:

```
$      J    # #   %
```

Notice the structure of this command:

```
$%      %
```

After the command “alias” we give alias a name followed immediately (no whitespace allowed) by an equals sign, followed immediately by a quoted string containing the meaning to be assigned to the name. After we define our alias, it can be used anywhere the shell would expect a command. Let's try it:

```
+&          6
5  5
```

We can also use the ,6 command again to see our alias:

```
!
I 5 T T  J
```

To remove an alias, the command is used, like so:

```
!
,6
```

While we purposefully avoided naming our alias with an existing command name, it is not uncommon to do so. This is often done to apply a commonly desired option to each invocation of a common command. For instance, we saw earlier how the command is often aliased to add color support:

```
                    !
        I       L  ,J
```

To see all the aliases defined in the environment, use the command without arguments. Here are some of the aliases defined by default on a Fedora system. Try and figure out what they all do:

```
    <LJ    <D      L  ,J
     LJ         L  ,J
     LJ       L  ,J
```

There is one tiny problem with defining aliases on the command line. They vanish when your shell session ends. In a later chapter, we will see how to add our own aliases to the files that establish the environment each time we log on, but for now, enjoy the fact that we have taken our first, albeit tiny, step into the world of shell programming!

Summing Up

Now that we have learned how to find the documentation for commands, go and look up the documentation for all the commands we have encountered so far. Study what additional options are available and try them out!

Further Reading

There are many online sources of documentation for Linux and the command line. Here are some of the best:

The 9 : % is a reference guide to the shell. It's still a reference work but contains examples and is easier to read than the man page.
http://www.gnu.org/software/bash/manual/bashref.html

The 9 *; contains answers to frequently asked questions regarding .
This list is aimed at intermediate to advanced users, but contains a lot of good information.
http://mywiki.wooledge.org/BashFAQ

The GNU Project provides extensive documentation for its programs, which form the core of the Linux command line experience. You can see a complete list here:
http://www.gnu.org/manual/manual.html

Wikipedia has an interesting article on man pages:
http://en.wikipedia.org/wiki/Man_page

, -

In this lesson we are going to unleash what may be the coolest feature of the command line. It's called +<) . The “I/O” stands for < and with this facility you can redirect the input and output of commands to and from files, as well as connect multiple commands together into powerful command . To show off this facility, we will introduce the following commands:

- Concatenate files
- Sort lines of text
A - Report or omit repeated lines
6 - Print lines matching a pattern
8 - Print newline, word, and byte counts for each file
- Output the first part of a file
- Output the last part of a file
- Read from standard input and write to standard output and files

Standard Input, Output, And Error

Many of the programs that we have used so far produce output of some kind. This output often consists of two types. First, we have the program's results; that is, the data the program is designed to produce, and second, we have status and error messages that tell us how the program is getting along. If we look at a command like , we can see that it displays its results and its error messages on the screen.

Keeping with the Unix theme of “everything is a file,” programs such as actually send their results to a special file called (often expressed as) and their status messages to another file called s ( ). By default, both standard output and standard error are linked to the screen and not saved into a disk file.

In addition, many programs take input from a facility called ( ) which is, by default, attached to the keyboard.

I/O redirection allows us to change where output goes and where input comes from. Normally, output goes to the screen and input comes from the keyboard, but with I/O redirection, we can change that.

Redirecting Standard Output

I/O redirection allows us to redefine where standard output goes. To redirect standard output to another file instead of the screen, we use the “>” redirection operator followed by the name of the file. Why would we want to do this? It's often useful to store the output of a command in a file. For example, we could tell the shell to send the output of the command to the file 6 < instead of the screen:

```
&
```

Here, we created a long listing of the 5 5 directory and sent the results to the file 6 < . Let's examine the redirected output of the command:

```
8 8 "  "+0!0! *$"+ $* $" "# $0  6 <
```

Good; a nice, large, text file. If we look at the file with , we will see that the file 6 < does indeed contain the results from our command:

```

```

Now, let's repeat our redirection test, but this time with a twist. We'll change the name of the directory to one that does not exist:

```
&
5 5 K  ,
```

We received an error message. This makes sense since we specified the non-existent directory 5 5 , but why was the error message displayed on the screen rather than being redirected to the file 6 < ? The answer is that the program does not send its error messages to standard output. Instead, like most well-written Unix programs, it sends its error messages to standard error. Since we only redirected standard output and not standard error, the error message was still sent to the screen. We'll see how

to redirect standard error in just a minute, but first, let's look at what happened to our output file:

```
8 8   "       $ *$"+ $* $" "# $!     6 <
```

The file now has zero length! This is because, when we redirect output with the “>” redirection operator, the destination file is always rewritten from the beginning. Since our command generated no results and only an error message, the redirection operation started to rewrite the file and then stopped because of the error, resulting in its truncation. In fact, if we ever need to actually truncate a file (or create a new, empty file) we can use a trick like this:

```
                    &
```

Simply using the redirection operator with no command preceding it will truncate an existing file or create a new, empty file.

So, how can we append redirected output to a file instead of overwriting the file from the beginning? For that, we use the “>>” redirection operator, like so:

```
                          &&
```

Using the “>>” operator will result in the output being appended to the file. If the file does not already exist, it is created just as though the “>” operator had been used. Let's put it to the test:

```
                          &&
                          &&
                          &&

8 8   "       #$/+/& *$"+ $* $" "# &#     6 <
```

We repeated the command three times resulting in an output file three times as large.

Redirecting Standard Error

Redirecting standard error lacks the ease of a dedicated redirection operator. To redirect

standard error we must refer to its . A program can produce output on any of several numbered file streams. While we have referred to the first three of these file streams as standard input, output and error, the shell references them internally as file descriptors 0, 1 and 2, respectively. The shell provides a notation for redirecting files using the file descriptor number. Since standard error is the same as file descriptor number 2, we can redirect standard error with this notation:

```
                                &
```

The file descriptor “2” is placed immediately before the redirection operator to perform the redirection of standard error to the file < .

Redirecting Standard Output And Standard Error To One File

There are cases in which we may wish to capture all of the output of a command to a single file. To do this, we must redirect both standard output and standard error at the same time. There are two ways to do this. First, the traditional way, which works with old versions of the shell:

```
                                &          &'
```

Using this method, we perform two redirections. First we redirect standard output to the file 6 < and then we redirect file descriptor 2 (standard error) to file descriptor one (standard output) using the notation *CY" .

---

2 " " The redirection of standard error must always occur redirecting standard output or it doesn't work. In the example above,

C 6 < *CY"

redirects standard error to the file 6 < , but if the order is changed to

*CY" C 6 <

standard error is directed to the screen.

Recent versions of provide a second, more streamlined method for performing this

combined redirection:

```
'&
```

In this example, we use the single notation YC to redirect both standard output and standard error to the file 6 < . You may also append the standard output and standard error streams to a single file like so:

```
'&&
```

Disposing Of Unwanted Output

Sometimes “silence is golden,” and we don't want output from a command, we just want to throw it away. This applies particularly to error and status messages. The system provides a way to do this by redirecting output to a special file called “/dev/null”. This file is a system device called a ! ! which accepts input and does nothing with it. To suppress error messages from a command, we do this:

```
&
```

> **In Unix Culture**
>
> The bit bucket is an ancient Unix concept and due to its universality, it has appeared in many parts of Unix culture. When someone says he/she is sending your comments to 5 5 , now you know what it means. For more examples, see the Wikipedia article on “/dev/null”.

Redirecting Standard Input

Up to now, we haven't encountered any commands that make use of standard input (actually we have, but we’ll reveal that surprise a little bit later), so we need to introduce one.

– Concatenate Files

The command reads one or more files and copies them to standard output like so:

```
(          )
```

In most cases, you can think of as being analogous to the)U:' command in DOS. You can use it to display files without paging, for example:

```

```

will display the contents of the file 6 < . is often used to display short text files. Since can accept more than one file as an argument, it can also be used to join files together. Say we have downloaded a large file that has been split into multiple parts (multimedia files are often split this way on Usenet), and we want to join them back together. If the files were named:

< 6 <$$" < 6 <$$* <<< < 6 <$%%

we could join them back together with this command:

```
     * &
```

Since wildcards always expand in sorted order, the arguments will be arranged in the correct order.

This is all well and good, but what does this have to do with standard input? Nothing yet, but let's try something else. What happens if we enter “cat” with no arguments:

```

```

Nothing happens, it just sits there like it's hung. It may seem that way, but it's really doing exactly what it's supposed to.

If is not given any arguments, it reads from standard input and since standard input is, by default, attached to the keyboard, it's waiting for us to type something! Try adding the following text and pressing Enter:

```
+  ,                    -!
```

Next, type a (i.e., hold down the Ctrl key and press “d”) to tell that it has reached (EOF) on standard input:

```
+  ,                    -!
)  A     8     6         B,   <
```

In the absence of filename arguments, copies standard input to standard output, so we see our line of text repeated. We can use this behavior to create short text files. Let's say that we wanted to create a file called “lazy_dog.txt” containing the text in our example. We would do this:

```
                        &   -!.
+  ,                    -!
```

Type the command followed by the text we want in to place in the file. Remember to type at the end. Using the command line, we have implemented the world's dumbest word processor! To see our results, we can use to copy the file to stdout again:

```
                         -!.
)  A     8     6         B,   <
```

Now that we know how accepts standard input, in addition to filename arguments, let's try redirecting standard input:

```
                        /   -!.
)  A     8     6         B,   <
```

Using the “<” redirection operator, we change the source of standard input from the keyboard to the file B,N < . We see that the result is the same as passing a single filename argument. This is not particularly useful compared to passing a filename argument, but it serves to demonstrate using a file as a source of standard input. Other commands make better use of standard input, as we shall soon see.

Before we move on, check out the man page for , as it has several interesting options.

Pipelines

The ability of commands to read data from standard input and send to standard output is utilized by a shell feature called . Using the pipe operator "|" (vertical bar), the standard output of one command can be into the standard input of another:

```
P
```

To fully demonstrate this, we are going to need some commands. Remember how we said there was one we already knew that accepts standard input? It's . We can use to display, page-by-page, the output of any command that sends its results to standard output:

```
0
```

This is extremely handy! Using this technique, we can conveniently examine the output of any command that produces standard output.

The Difference Between > and |

At first glance, it may be hard to understand the redirection performed by the pipeline operator P versus the redirection operator C. Simply put, the redirection operator connects a command with a file while the pipeline operator connects the output of one command with the input of a second command.

C

P

A lot of people will try the following when they are learning about pipelines, "just to see what happens."

C

Answer: Sometimes something really bad.

Here is an actual example submitted by a reader who was administering a Linux-based server appliance. As the superuser, he did this:

Z 5 5

Z C

The first command put him in the directory where most programs are stored and the second command told the shell to overwrite the file with the output of the command. Since the 5 5 directory already contained a file named "less" (the program), the second command overwrote the program file with the text from thus destroying the program on his system.

The lesson here is that the redirection operator silently creates or overwrites files, so you need to treat it with a lot of respect.

Filters

Pipelines are often used to perform complex operations on data. It is possible to put several commands together into a pipeline. Frequently, the commands used this way are referred to as . Filters take input, change it somehow and then output it. The first one we will try is . Imagine we wanted to make a combined list of all of the executable programs in 5 and 5 5 , put them in sorted order and view it:

```
0 0
```

Since we specified two directories (5 and 5 5), the output of would have consisted of two sorted lists, one for each directory. By including in our pipeline, we changed the data to produce a single, sorted list.

uniq - Report Or Omit Repeated Lines

The A command is often used in conjunction with . A accepts a sorted list of data from either standard input or a single filename argument (see the A man page for details) and, by default, removes any duplicates from the list. So, to make sure our list has no duplicates (that is, any programs of the same name that appear in both the 5 and 5 5 directories) we will add A to our pipeline:

```
0 0 ,0
```

In this example, we use A to remove any duplicates from the output of the command. If we want to see the list of duplicates instead, we add the “-d” option to A like so:

```
0 0 , 0
```

8 – Print Line, Word, And Byte Counts

The 8 (word count) command is used to display the number of lines, words, and bytes contained in files. For example:

```
0%$* +&#++ #$/+/& 6 <
```

In this case it prints out three numbers: lines, words, and bytes contained in 6 < . Like our previous commands, if executed without command line arguments, 8 accepts standard input. The “-l” option limits its output to only report lines. Adding it to a pipeline is a handy way to count things. To see the number of items we have in our sorted list, we can do this:

```
0 0 , 0
*0*!
```

6 – Print Lines Matching A Pattern

6 is a powerful program used to find text patterns within files. It's used like this:

```
6
```

When 6 encounters a “pattern” in the file, it prints out the lines containing it. The patterns that 6 can match can be very complex, but for now we will concentrate on simple text matches. We'll cover the advanced patterns, called in a later chapter.

Let's say we wanted to find all the files in our list of programs that had the word “zip” embedded in the name. Such a search might give us an idea of some of the programs on

our system that had something to do with file compression. We would do this:

```
                                0    0   , 0    -
  B 6*
 B 6*
  B 6
 B 6
  B 6
B 6
B 6
B 6   6
B 6
B 6
B 6 6
```

There are a couple of handy options for 6 : “-i” which causes 6 to ignore case when performing the search (normally searches are case sensitive) and “-v” which tells 6 to only print lines that do not match the pattern.

/ – Print First / Last Part Of Files

Sometimes you don't want all the output from a command. You may only want the first few lines or the last few lines. The command prints the first ten lines of a file and the command prints the last ten lines. By default, both commands print ten lines of text, but this can be adjusted with the “-n” option:

```
                              "
   /&/&%+
 8      "            /"/"+ *$$0 "* $# $! #!
 8      "             !*&$ *$$0 "* $% "/ /% &""  66
 8      "           """"*0+ *$$0 "" *+ "& *0  *6
 8      "            *#/+! *$$+ "$ $+ *$ "+  #*
                              "
 8      "             #*/& *$$0 $+ *0 "$ #+ B  8
 8      "              +%" *$$# $% "$ $& *" B     *6  <6,
 8      "              %/$ *$$0 "" $" "* */ B     *6  <6,
 8      "              %/$ *$$0 "" $" "* */ B     *6  <6,
 8  8  8  "               + *$"+ $" /" $# ** B       C
```

These can be used in pipelines as well:

```
                               0       "
```

```
B  8
B     *6  <6,
B     *6  <6,
B     *6  <6,
B
```

has an option which allows you to view files in real-time. This is useful for watching the progress of log files as they are being written. In the following example, we will look at the file in 5 5 (or the 5 5 5 , file if is missing). Superuser privileges are required to do this on some Linux distributions, since the 5 5 5 file may contain security information:

```
    ! "/ &$ $#  8  &         9R :3 1      "%*<"+!<"<"
    ! "/ &$ $#  8  &               "%*<"+!<"<&       8
"+#*       <
    ! "/ ## /*  8  &       /%#/   5   5K ( &5         6
    "%*<"+!<"<$5*&     8  0<
"%*<"+!<"<$5*&> 8  0<
    ! "& $0 /0  8  &         9R :Q'[2'()      $    "%*<"+!<"<"
6    +0
    ! "& $0 /0  8  &         9R :3 1      "%*<"+!<"<"
    ! "& $0 /0  8  &               "%*<"+!<"<&       8
"00"       <
    ! "& $% #+  8  &       /&+!   9      5   5   > (-3Q)
:         3         ! (   N)   N:                  */0    */+
    ! "& "$ /0  8  &       /%#/   5   5K ( &5         6
    "%*<"+!<"<$5*&     8  0<
"%*<"+!<"<$5*&> 8  0<
    ! "& *# $0  8  &     M6  N    O *%*/&          6
   , M   L$O
    ! "& *# /+  8  &   M6  N    O *%*0%          6
     ,   M   L#$$O
```

Using the “-f” option, continues to monitor the file and when new lines are appended, they immediately appear on the display. This continues until you type .

– Read From Stdin And Output To Stdout And Files

In keeping with our plumbing metaphor, Linux provides a command called which creates a “tee” fitting on our pipe. The program reads standard input and copies it to both standard output (allowing the data to continue down the pipeline) and to one or more files. This is useful for capturing a pipeline's contents at an intermediate stage of processing. Here we repeat one of our earlier examples, this time including to capture the

entire directory listing to the file `<` before `6` filters the pipeline's contents:

```
                          0       0    -
  B 6*
 B 6*
  B 6
 B 6
  B 6
B 6
B 6
B 6  6
B 6
B 6
B 6 6
```

Summing Up

As always, check out the documentation of each of the commands we have covered in this chapter. We have only seen their most basic usage. They all have a number of interesting options. As we gain Linux experience, we will see that the redirection feature of the command line is extremely useful for solving specialized problems. There are many commands that make use of standard input and output, and almost all command line programs use standard error to display their informative messages.

Linux Is About Imagination

When I am asked to explain the difference between Windows and Linux, I often use a toy analogy.

Windows is like a Game Boy. You go to the store and buy one all shiny new in the box. You take it home, turn it on and play with it. Pretty graphics, cute sounds. After a while though, you get tired of the game that came with it so you go back to the store and buy another one. This cycle repeats over and over. Finally, you go back to the store and say to the person behind the counter, “I want a game that does this!” only to be told that no such game exists because there is no “market demand” for it. Then you say, “But I only need to change this one thing!” The person behind the counter says you can't change it. The games are all sealed up in their cartridges. You discover that your toy is limited to the games that others have decided that you need and no more.

> Linux, on the other hand, is like the world's largest Erector Set. You open it up and it's just a huge collection of parts. A lot of steel struts, screws, nuts, gears, pulleys, motors, and a few suggestions on what to build. So you start to play with it. You build one of the suggestions and then another. After a while you discover that you have your own ideas of what to make. You don't ever have to go back to the store, as you already have everything you need. The Erector Set takes on the shape of your imagination. It does what you want.
>
> Your choice of toys is, of course, a personal thing, so which toy would you find more satisfying?

(

In this chapter we are going to look at some of the “magic” that occurs on the command line when we press the enter key. While we will examine several interesting and complex features of the shell, we will do it with just one new command:

– Display a line of text

Expansion

Each time we type a command and press the enter key, performs several processes upon the text before it carries out our command. We have seen a couple of cases of how a simple character sequence, for example “*”, can have a lot of meaning to the shell. The process that makes this happen is called . With expansion, we enter something and it is expanded into something else before the shell acts upon it. To demonstrate what we mean by this, let's take a look at the command. is a shell builtin that performs a very simple task. It prints out its text arguments on standard output:

```

```

That's pretty straightforward. Any argument passed to gets displayed. Let's try another example:

```
9  69      6<  -  :   :  )6
;
```

So what just happened? Why didn't print “*”? As we recall from our work with wildcards, the “*” character means match any characters in a filename, but what we didn't see in our original discussion was how the shell does that. The simple answer is that the shell expands the “*” into something else (in this instance, the names of the files in the current working directory) before the command is executed. When the enter key is

pressed, the shell automatically expands any qualifying characters on the command line before the command is carried out, so the command never saw the “*”, only its expanded result. Knowing this, we can see that behaved as expected.

Pathname Expansion

The mechanism by which wildcards work is called . If we try some of the techniques that we employed in our earlier chapters, we will see that they are really expansions. Given a home directory that looks like this:

```
9  6     6 <  :    ) 6
9    -       :    ;
```

we could carry out the following expansions:

```
                    1
9  6 9
```

and:

```
9    :    ) 6   ;
```

or even:

```
                    ((2   2))
9  6 9    -  :    :   ) 6   ;
```

and looking beyond our home directory:

```
5 5    5   5 5  5
```

Pathname Expansion Of Hidden Files

As we know, filenames that begin with a period character are hidden. Pathname expansion also respects this behavior. An expansion such as:

D

does not reveal hidden files.

It might appear at first glance that we could include hidden files in an expansion by starting the pattern with a leading period, like this:

<D

It almost works. However, if we examine the results closely, we will see that the names “.” and “..” will also appear in the results. Since these names refer to the current working directory and its parent directory, using this pattern will likely produce an incorrect result. We can see this if we try the command:

<D P

To better perform pathname expansion in this situation, we have to employ a more specific pattern:

< F< D

This pattern expands into every filename that begins with a period, does not include a second period, and followed by any other characters. This will work correctly with most hidden files (though it still won't include filenames with multiple leading periods). The command with the 3 option (“almost all”) will provide a correct listing of hidden files:

3

Tilde Expansion

As we may recall from our introduction to the command, the tilde character (“~”) has a special meaning. When used at the beginning of a word, it expands into the name of the home directory of the named user, or if no user is named, the home directory of the current user:

```
5  5
```

If user “foo” has an account, then:

```
5  5
```

Arithmetic Expansion

The shell allows arithmetic to be performed by expansion. This allows us to use the shell prompt as a calculator:

```
                    344 5 66
&
```

Arithmetic expansion uses the form:

MM OO

where is an arithmetic expression consisting of values and arithmetic operators.

Arithmetic expansion only supports integers (whole numbers, no decimals), but can perform quite a number of different operations. Here are a few of the supported operators:

! 4 *)

Operator	Description
7	Addition
	Subtraction
D	Multiplication
5	Division (but remember, since expansion only supports integer arithmetic, results are integers).
4	Modulo, which simply means, “ remainder.”
DD	Exponentiation

Spaces are not significant in arithmetic expressions and expressions may be nested. For example, to multiply 5 squared by 3:

```
                    344344"  66  66
0#
```

Single parentheses may be used to group multiple subexpressions. With this technique, we can rewrite the example above and get the same result using a single expansion instead of two:

```
                        3444"   6    66
0#
```

Here is an example using the division and remainder operators. Notice the effect of integer division:

```
                        7          !     ,    344"  66
         , 8   A     *
                             344"8 66
8    "        <
```

Arithmetic expansion is covered in greater detail in Chapter 34.

Brace Expansion

Perhaps the strangest expansion is called ! . With it, you can create multiple text strings from a pattern containing braces. Here's an example:

```
                        7     9:;<;=> <
    3 G         G G          G
```

Patterns to be brace expanded may contain a leading portion called a ! and a trailing portion called a . The brace expression itself may contain either a comma-separated list of strings, or a range of integers or single characters. The pattern may not contain embedded whitespace. Here is an example using a range of integers:

```
                        ?     .9   ">
K     N" K     N* K     N/ K     N& K     N#
```

Integers may also be 8 like so:

```
                        9*    ">
$" $* $/ $& $# $+ $0 $! $% "$ "" "* "/ "& "#
```

```
                              9**    ">
$$" $$* $$/ $$& $$# $$+ $$0 $$! $$% $"$ $"" $"* $"/ $"& $"#
```

A range of letters in reverse order:

```
                              9@  :>
H U V . ; 2 ) ( Q [ : S K - = 1   @ R ?  ' 9   G 3
```

Brace expansions may be nested:

```
                              9:9 ; >;<9 ;A>>
 3"   3*   G/   G&
```

So what is this good for? The most common application is to make lists of files or directories to be created. For example, if we were photographers and had a large collection of images that we wanted to organize into years and months, the first thing we might do is create a series of directories named in numeric “Year-Month” format. This way, the directory names will sort in chronological order. We could type out a complete list of directories, but that's a lot of work and it's error-prone too. Instead, we could do this:

```
                               B
                              B
        :                              9 **C   **D> 9*     >
        :
*$$0 $"  *$$0 $0  *$$! $"  *$$! $0  *$$% $"  *$$% $0
*$$0 $*  *$$0 $!  *$$! $*  *$$! $!  *$$% $*  *$$% $!
*$$0 $/  *$$0 $%  *$$! $/  *$$! $%  *$$% $/  *$$% $%
*$$0 $&  *$$0 "$  *$$! $&  *$$! "$  *$$% $&  *$$% "$
*$$0 $#  *$$0 ""  *$$! $#  *$$! ""  *$$% $#  *$$% ""
*$$0 $+  *$$0 "*  *$$! $+  *$$! "*  *$$% $+  *$$% "*
```

Pretty slick!

Parameter Expansion

We're only going to touch briefly on parameter expansion in this chapter, but we'll be covering it extensively later. It's a feature that is more useful in shell scripts than directly on the command line. Many of its capabilities have to do with the system's ability to store small chunks of data and to give each chunk a name. Many such chunks, more properly

called ! , are available for your examination. For example, the variable named "USER" contains your username. To invoke parameter expansion and reveal the contents of 2('Q you would do this:

```
3EFGH
```

To see a list of available variables, try this:

```
0
```

You may have noticed that with other types of expansion, if you mistype a pattern, the expansion will not take place and the command will simply display the mistyped pattern. With parameter expansion, if you misspell the name of a variable, the expansion will still take place, but will result in an empty string:

```
3FEGH
```

Command Substitution

Command substitution allows us to use the output of a command as an expansion:

```
34 6
9 6 9 6 < - : : ) 6
;
```

One of my favorites goes something like this:

```
34 6
8 " 0"#"+ *$$0 "* $# $! #! 5 5 6
```

Here we passed the results of 8 6 as an argument to the command, thereby getting the listing of of the 6 program without having to know its full pathname. We are not limited to just simple commands. Entire pipelines can be used (only partial output

shown):

```
                          34            0     -  6
5   5   5   B 6*       ,           I B 6*J
5   5   5 B 6*        '=  /*    =(G         > @     !$/!+>
      " M(U(;O>  ,       ,       M             O>
?K25=     *<+<%>     66
5   5   5 B 6*        '=  /*    =(G         > @     !$/!+>
      " M(U(;O>  ,       ,       M             O>
?K25=     *<+<%>     66
5   5   5   B 6       '=  /*    =(G         > @     !$/!+>
      " M(U(;O>  ,       ,       M             O>
?K25=     *<+<%>     66
5   5   5 6  B 6      G             6
5   5   5   B 6       ,           I<<5<<5   5   B 6J
5   5   5 B 6         ,           I<<5<<5   5 B 6J
5   5   5 B 6         ,           I     J
```

In this example, the results of the pipeline became the argument list of the com-mand.

There is an alternate syntax for command substitution in older shell programs which is also supported in . It uses ! = instead of the dollar sign and parentheses:

```
                              I       I
 8       "          0"#"+ *$$0 "* $# $! #! 5   5 6
```

Quoting

Now that we've seen how many ways the shell can perform expansions, it's time to learn how we can control it. Take for example:

```

```

or:

```
                              +          3 ** **
)         $$<$$
```

In the first example, by the shell removed extra whitespace from the command's list of arguments. In the second example, parameter expansion substituted an empty string for the value of “$1” because it was an undefined variable. The shell provides a mechanism called = to selectively suppress unwanted expansions.

Double Quotes

The first type of quoting we will look at is ! = . If we place text inside double quotes, all the special characters used by the shell lose their special meaning and are treated as ordinary characters. The exceptions are “$”, “\” (backslash), and “`” (backquote). This means that word-splitting, pathname expansion, tilde expansion, and brace expansion are suppressed, but parameter expansion, arithmetic expansion, and command substitution are still carried out. Using double quotes, we can cope with filenames containing embedded spaces. Say we were the unfortunate victim of a file called 8 8 < . If we tried to use this on the command line, word-splitting would cause this to be treated as two separate arguments rather than the desired single argument:

```
        8 K          ,
        8 < K          ,
```

By using double quotes, we stop the word-splitting and get the desired result; further, we can even repair the damage:

```
                      J      J
 8 8  "      "! *$"+ $* *$ "/ $/ 8 8 <
                    J      J .
```

There! Now we don't have to keep typing those pesky double quotes.

Remember, parameter expansion, arithmetic expansion, and command substitution still take place within double quotes:

```
                        J3EFGH 344 5 66 34  6J
  &       , *$"+
( - ) . )  (
          " *
 / & # + 0 ! %
"$ "" "* "/ "& "# "+
"0 "! "% *$ *" ** */
*& *# *+ *0 *! *%
```

We should take a moment to look at the effect of double quotes on command substitution. First let's look a little deeper at how word splitting works. In our earlier example, we saw how word-splitting appears to remove extra spaces in our text:

```

```

By default, word-splitting looks for the presence of spaces, tabs, and newlines (linefeed characters) and treats them as between words. This means that unquoted spaces, tabs, and newlines are not considered to be part of the text. They only serve as separators. Since they separate the words into different arguments, our example command line contains a command followed by four distinct arguments. If we add double quotes:

```
                    J          J

```

word-splitting is suppressed and the embedded spaces are not treated as delimiters, rather they become part of the argument. Once the double quotes are added, our command line contains a command followed by a single argument.

The fact that newlines are considered delimiters by the word-splitting mechanism causes an interesting, albeit subtle, effect on command substitution. Consider the following:

```
                          34  6
     , *$"+ ( - ) . )    (  " * / & # + 0 ! % "$ "" "* "/ "&
"# "+ "0 "! "% *$ *" ** */ *& *# *+ *0 *! *%
                          J34  6J
       , *$"+
( - ) . )   (
          "  *
 / & # + 0 ! %
"$ "" "* "/ "& "# "+
"0 "! "% *$ *" ** */
*& *# *+ *0 *! *%
```

In the first instance, the unquoted command substitution resulted in a command line containing 38 arguments. In the second, a command line with one argument that includes the embedded spaces and newlines.

Single Quotes

If we need to suppress expansions, we use = . Here is a comparison of unquoted, double quotes, and single quotes:

```
                                    9 ; > 34      6 344 5 66 3EFGH
   5   5 5    6 <          &
                              J          9 ; > 34      6 344 5 66 3EFGHJ
    5D<   \ > ]    &
                              %          9 ; > 34      6 344 5 66 3EFGH%
    5D<   \ > ]  M      O  MM*7*OO  2('Q
```

As we can see, with each succeeding level of quoting, more and more of the expansions are suppressed.

Escaping Characters

Sometimes we only want to quote a single character. To do this, we can precede a character with a backslash, which in this context is called the . Often this is done inside double quotes to selectively prevent an expansion:

```
                              J+                3EFGH   2 K3" **J
)                    #<$$
```

It is also common to use escaping to eliminate the special meaning of a character in a filename. For example, it is possible to use characters in filenames that normally have special meaning to the shell. These would include “$”, “!”, “&”, “ ”, and others. To include a special character in a filename you can do this:

```
                              K'          .
```

To allow a backslash character to appear, escape it by typing “\\”. Note that within single quotes, the backslash loses its special meaning and is treated as an ordinary character.

Backslash Escape Sequences

In addition to its role as the escape character, the backslash is also used as part of a notation to represent certain special characters called . The first 32 characters in the ASCII coding scheme are used to transmit commands to teletype-like devices. Some of these codes are familiar (tab, backspace, linefeed, and carriage return), while others are not (null, end-of-transmission, and acknowledge).

Escape Sequence	Meaning
^	Bell (“Alert” - causes the computer to beep)
^	Backspace
^	Newline. On Unix-like systems, this produces a linefeed.
^	Carriage return
^	Tab

The table above lists some of the common backslash escape sequences. The idea behind this representation using the backslash originated in the C programming language and has been adopted by many others, including the shell.

Adding the “-e” option to will enable interpretation of escape sequences. You may also place them inside J J . Here, using the 6 command, a simple program that just waits for the specified number of seconds and then exits, we can create a primitive countdown timer:

```
*#      J+  %   K J
```

We could also do this:

```
*#    J+  %   J 3%K %
```

Summing Up

As we move forward with using the shell, we will find that expansions and quoting will be used with increasing frequency, so it makes sense to get a good understanding of the way they work. In fact, it could be argued that they are the most important subjects to learn about the shell. Without a proper understanding of expansion, the shell will always be a source of mystery and confusion, and much of its potential power wasted.

Further Reading

The man page has major sections on both expansion and quoting which cover these topics in a more formal manner.

The 9 : % also contains chapters on expansion and quoting:
http://www.gnu.org/software/bash/manual/bashref.html

. (/ # +

I often kiddingly describe Unix as “the operating system for people who like to type.” Of course, the fact that it even has a command line is a testament to that. But command line users don't like to type much. Why else would so many commands have such short names like 6 , , , and ? In fact, one of the most cherished goals of the command line is laziness; doing the most work with the fewest number of keystrokes. Another goal is never having to lift your fingers from the keyboard, never reaching for the mouse. In this chapter, we will look at features that make keyboard use faster and more efficient.

The following commands will make an appearance:

– Clear the screen

, – Display the contents of the history list

Command Line Editing

uses a library (a shared collection of routines that different programs can use) called : to implement command line editing. We have already seen some of this. We know, for example, that the arrow keys move the cursor but there are many more features. Think of these as additional tools that we can employ in our work. It’s not important to learn all of them, but many of them are very useful. Pick and choose as desired.

---

2 6 Some of the key sequences below (particularly those which use the 3 key) may be intercepted by the GUI for other functions. All of the key sequences should work properly when using a virtual console.

Cursor Movement

The following table lists the keys used to move the cursor:

! 5 %

Key	Action
	Move cursor to the beginning of the line.
	Move cursor to the end of the line.
	Move cursor forward one character; same as the right arrow key.
	Move cursor backward one character; same as the left arrow key.
3	Move cursor forward one word.
3	Move cursor backward one word.
	Clear the screen and move the cursor to the top left corner. The command does the same thing.

Modifying Text

Table 8-2 lists keyboard commands that are used to edit characters on the command line.

! 5 " 1

Key	Action
	Delete the character at the cursor location
	Transpose (exchange) the character at the cursor location with the one preceding it.
3	Transpose the word at the cursor location with the one preceding it.
3	Convert the characters from the cursor location to the end of the word to lowercase.
3	Convert the characters from the cursor location to the end of the word to uppercase.

Cutting And Pasting (Killing And Yanking) Text

The Readline documentation uses the terms and to refer to what we would commonly call cutting and pasting. Items that are cut are stored in a buffer called the .

! 5 (* 7

Key	Action
	Kill text from the cursor location to the end of line.
	Kill text from the cursor location to the beginning of the line.
3	Kill text from the cursor location to the end of the current word.
3 G 6	Kill text from the cursor location to the beginning of the current word. If the cursor is at the beginning of a word, kill the previous word.
,	Yank text from the kill-ring and insert it at the cursor location.

The Meta Key

If you venture into the Readline documentation, which can be found in the READLINE section of the man page, you will encounter the term “meta key.” On modern keyboards this maps to the 3 key but it wasn't always so.

Back in the dim times (before PCs but after Unix) not everybody had their own computer. What they might have had was a device called a . A terminal was a communication device that featured a text display screen and a keyboard and just enough electronics inside to display text characters and move the cursor around. It was attached (usually by serial cable) to a larger computer or the communication network of a larger computer. There were many different brands of terminals and they all had different keyboards and display feature sets. Since they all tended to at least understand ASCII, software developers wanting portable applications wrote to the lowest common denominator. Unix systems have a very elaborate way of dealing with terminals and their different display features. Since the developers of Readline could not be sure of the presence of a dedicated extra control key, they invented one and called it “meta.” While the 3 key serves as the meta key on modern keyboards, you can also press and release the ' key to get the same effect as holding down the 3 key if you're still using a terminal (which you can still do in Linux!).

Completion

Another way that the shell can help you is through a mechanism called . Completion occurs when you press the tab key while typing a command. Let's see how this

works. Given a home directory that looks like this:

```
9   6     6 <   :     ) 6      ;
9     -       :
```

Try typing the following but 8 (G :

```

```

Now press the tab key:

```

```

See how the shell completed the line for you? Let's try another one. Again, don't press
' :

```
                    1
```

Press tab:

```
                    1
```

No completion, just a beep. This happened because “D” matches more than one entry in the directory. For completion to be successful, the “clue” you give it has to be unambiguous. If we go further:

```
                    1
```

Then press tab:

```
                    1
```

The completion is successful.

While this example shows completion of pathnames, which is its most common use, completion will also work on variables (if the beginning of the word is a “$”), user names (if the word begins with “~”), commands (if the word is the first word on the line) and hostnames (if the beginning of the word is “@”). Hostname completion only works for hostnames listed in 5 5 .

There are a number of control and meta key sequences that are associated with completion:

! 5 -

Key	Action
3 E	Display list of possible completions.) ! ! & '
3 D	Insert all possible completions. This is useful when you want to use more than one possible match.

There quite a few more that I find rather obscure. You can see a list in the man page under “READLINE”.

Programmable Completion

Recent versions of have a facility called ! . Programmable completion allows you (or more likely, your distribution provider) to add additional completion rules. Usually this is done to add support for specific applications. For example, it is possible to add completions for the option list of a command or match particular file types that an application supports. Ubuntu has a fairly large set defined by default. Programmable completion is implemented by shell functions, a kind of mini shell script that we will cover in later chapters. If you are curious, try:

0

and see if you can find them. Not all distributions include them by default.

Using History

As we discovered in Chapter 1, maintains a history of commands that have been entered. This list of commands is kept in your home directory in a file called

< N , . The history facility is a useful resource for reducing the amount of typing you have to do, especially when combined with command line editing.

Searching History

At any time, we can view the contents of the history list by:

```
! 0
```

By default, bash stores the last 500 commands we have entered, though most modern distributions set this value to 1000. We will see how to adjust this value in a later chapter. Let's say we want to find the commands we used to list 5 5 . One way we could do this:

```
! 0
```

And let's say that among our results we got a line containing an interesting command like this:

```
!!  5 5 C  6 <
```

The number “88” is the line number of the command in the history list. We could use this immediately using another type of expansion called . To use our discovered line we could do this:

```
LMM
```

will expand “!88” into the contents of the eighty-eighth line in the history list. There are other forms of history expansion that we will cover a little later.

also provides the ability to search the history list incrementally. This means that we can tell to search the history list as we enter characters, with each additional character further refining our search. To start incremental search press followed by the text you are looking for. When you find it, you can either press ' to execute the command or press to copy the line from the history list to the current command line. To find the next occurrence of the text (moving “up” the history list), press again. To quit searching, press either or . Here we see it in action:

First press :

Immediately, the search returns our result. With our result, we can execute the command by pressing ' , or we can copy the command to our current command line for further editing by pressing . Let's copy it. Press :

Our shell prompt returns and our command line is loaded and ready for action!

The table below lists some of the keystrokes used to manipulate the history list:

Key	Action
6	Move to the previous history entry. Same action as the up arrow.
	Move to the next history entry. Same action as the down arrow.
3 X	Move to the beginning (top) of the history list.
3 C	Move to the end (bottom) of the history list, i.e., the current command line.
	Reverse incremental search. Searches incrementally from the current command line up the history list.
3 6	Reverse search, non-incremental. With this key, type in the search string and press enter before the search is performed.

3	Forward search, non-incremental.
	Execute the current item in the history list and advance to the next one. This is handy if you are trying to re-execute a sequence of commands in the history list.

History Expansion

The shell offers a specialized type of expansion for items in the history list by using the "!" character. We have already seen how the exclamation point can be followed by a number to insert an entry from the history list. There are a number of other expansion features:

! 5 3 , 1

Sequence	Action
FF	Repeat the last command. It is probably easier to press up arrow and enter.
F	Repeat history list item ! .
F	Repeat last history list item starting with string.
FE	Repeat last history list item containing string.

I would caution against using the "!string" and "!?string" forms unless you are absolutely sure of the contents of the history list items.

There are many more elements available in the history expansion mechanism, but this subject is already too arcane and our heads may explode if we continue. The HISTORY EXPANSION section of the man page goes into all the gory details. Feel free to explore!

In addition to the command history feature in , most Linux distributions include a program called 6 that can be used to record an entire shell session and store it in a file. The basic syntax of the command is:

6

where is the name of the file used for storing the recording. If no file is specified, the file ,6 6 is used. See the 6 man page for a complete list of the program's options and features.

Summing Up

In this chapter we have covered of the keyboard tricks that the shell provides to help hardcore typists reduce their workloads. I suspect that as time goes by and you become more involved with the command line, you will refer back to this chapter to pick up more of these tricks. For now, consider them optional and potentially helpful.

Further Reading

The Wikipedia has a good article on computer terminals:
http://en.wikipedia.org/wiki/Computer_terminal

1 2 $

Operating systems in the Unix tradition differ from those in the MS-DOS tradition in that they are not only systems, but also systems, as well.

What exactly does this mean? It means that more than one person can be using the computer at the same time. While a typical computer will likely have only one keyboard and monitor, it can still be used by more than one user. For example, if a computer is attached to a network or the Internet, remote users can log in via (secure shell) and operate the computer. In fact, remote users can execute graphical applications and have the graphical output appear on a remote display. The X Window System supports this as part of its basic design.

The multiuser capability of Linux is not a recent "innovation," but rather a feature that is deeply embedded into the design of the operating system. Considering the environment in which Unix was created, this makes perfect sense. Years ago, before computers were "personal," they were large, expensive, and centralized. A typical university computer system, for example, consisted of a large central computer located in one building and terminals which were located throughout the campus, each connected to the large central computer. The computer would support many users at the same time.

In order to make this practical, a method had to be devised to protect the users from each other. After all, the actions of one user could not be allowed to crash the computer, nor could one user interfere with the files belonging to another user.

In this chapter we are going to look at this essential part of system security and introduce the following commands:

– Display user identity

– Change a file's mode

– Set the default file permissions

– Run a shell as another user

– Execute a command as another user

8 – Change a file's owner

6 – Change a file's group ownership

6 8 – Change a user's password

Owners, Group Members, And Everybody Else

When we were exploring the system back in Chapter 3, we may have encountered a problem when trying to examine a file such as 5 5 8 :

```
5 5 8 > 6
5 5 8 :
```

The reason for this error message is that, as regular users, we do not have permission to read this file.

In the Unix security model, a user may files and directories. When a user owns a file or directory, the user has control over its access. Users can, in turn, belong to a consisting of one or more users who are given access to files and directories by their owners. In addition to granting access to a group, an owner may also grant some set of access rights to everybody, which in Unix terms is referred to as the . To find out information about your identity, use the command:

```
L#$$M O L#$$M O 6 L#$$M O
```

Let's look at the output. When user accounts are created, users are assigned a number called a +. or which is then, for the sake of the humans, mapped to a username. The user is assigned a +. or and may belong to additional groups. The above example is from a Fedora system. On other systems, such as Ubuntu, the output may look a little different:

```
L"$$$M O L"$$$M O
6 L&M O>*$M O>*&M O>*#M 66,O>*%M O>/$M 6O>&&M
O>&+M6 O>"$!M 6 O>""&M O>"$$$M O
```

As we can see, the uid and gid numbers are different. This is simply because Fedora starts its numbering of regular user accounts at 500, while Ubuntu starts at 1000. We can also

see that the Ubuntu user belongs to a lot more groups. This has to do with the way Ubuntu manages privileges for system devices and services.

So where does this information come from? Like so many things in Linux, from a couple of text files. User accounts are defined in the 5 56 8 file and groups are defined in the 5 5 6 file. When user accounts and groups are created, these files are modified along with 5 5 8 which holds information about the user's password. For each user account, the 5 56 8 file defines the user (login) name, uid, gid, the account's real name, home directory, and login shell. If we examine the contents of 5 56 8 and 5 5 6 , we notice that besides the regular user accounts, there are accounts for the superuser (uid 0) and various other system users.

In the next chapter, when we cover processes, we will see that some of these other "users" are, in fact, quite busy.

While many Unix-like systems assign regular users to a common group such as "users", modern Linux practice is to create a unique, single-member group with the same name as the user. This makes certain types of permission assignment easier.

Reading, Writing, And Executing

Access rights to files and directories are defined in terms of read access, write access, and execution access. If we look at the output of the command, we can get some clue as to how this is implemented:

```
                    &

 8 8   "       $ *$"+ $/ $+ "& #*   <
```

The first ten characters of the listing are the ! . The first of these characters is the . Here are the file types you are most likely to see (there are other, less common types too):

! 6

Attribute	File Type
	A regular file.
	A directory.
	A symbolic link. Notice that with symbolic links, the remaining file attributes are always "rwxrwxrwx" and are dummy values. The real file attributes are those of the file the symbolic link points to.

	A . This file type refers to a device that handles data as a stream of bytes, such as a terminal or modem.
	A ! . This file type refers to a device that handles data in blocks, such as a hard drive or CD-ROM drive.

The remaining nine characters of the file attributes, called the represent the read, write, and execute permissions for the file's owner, the file's group owner, and everybody else:

Owner	Group	World
8	8	8

When set, the , 8, and mode attributes have the following effect on files and directories:

! 6 " 7 * !

Attribute	Files	Directories
	Allows a file to be opened and read.	Allows a directory's contents to be listed if the execute attribute is also set.
8	Allows a file to be written to or truncated, however this attribute does not allow files to be renamed or deleted. The ability to delete or rename files is determined by directory attributes.	Allows files within a directory to be created, deleted, and renamed if the execute attribute is also set.
	Allows a file to be treated as a program and executed. Program files written in scripting languages must also be set as readable to be executed.	Allows a directory to be entered, e.g., cd .

Here are some examples of file attribute settings:

! 6 (7 * ! 1

File Attributes	Meaning
8	A regular file that is readable, writable, and executable by the file's owner. No one else has any access.
8	A regular file that is readable and writable by the file's owner. No one else has any access.
8	A regular file that is readable and writable by the file's owner. Members of the file's owner group may read the file. The file is world-readable.
8	A regular file that is readable, writable, and executable by the file's owner. The file may be read and executed by everybody else.
8 8	A regular file that is readable and writable by the file's owner and members of the file's group owner only.
8 8 8	A symbolic link. All symbolic links have “dummy” permissions. The real permissions are kept with the actual file pointed to by the symbolic link.
8 8	A directory. The owner and the members of the owner group may enter the directory and, create, rename and remove files within the directory.
8	A directory. The owner may enter the directory and create, rename and delete files within the directory. Members of the owner group may enter the directory but cannot create, delete or rename files.

– Change File Mode

To change the mode (permissions) of a file or directory, the command is used. Be aware that only the file’s owner or the superuser can change the mode of a file or directory. supports two distinct ways of specifying mode changes: octal number representation, or symbolic representation. We will cover octal number representation first.

What The Heck Is Octal?

) (base 8), and its cousin, (base 16) are number systems often used to express numbers on computers. We humans, owing to the fact that we (or at least most of us) were born with ten fingers, count using a base 10 number system. Computers, on the other hand, were born with only one finger and thus do all all their counting in ! (base 2). Their number system only has two numerals, 0 and 1. So in binary, counting looks like this:

0, 1, 10, 11, 100, 101, 110, 111, 1000, 1001, 1010, 1011...

In octal, counting is done with the numerals zero through seven, like so:

0, 1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14, 15, 16, 17, 20, 21...

Hexadecimal counting uses the numerals zero through nine plus the letters “A” through “F”:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10, 11, 12, 13...

While we can see the sense in binary (since computers only have one finger), what are octal and hexadecimal good for? The answer has to do with human convenience. Many times, small portions of data are represented on computers as ! . Take for example an RGB color. On most computer displays, each pixel is composed of three color components: eight bits of red, eight bits of green, and eight bits of blue. A lovely medium blue would be a 24 digit number:

010000110110111111001101

How would you like to read and write those kinds of numbers all day? I didn't think so. Here's where another number system would help. Each digit in a hexadecimal number represents four digits in binary. In octal, each digit represents three binary digits. So our 24 digit medium blue could be condensed down to a six digit hexadecimal number:

436FCD

Since the digits in the hexadecimal number “line up” with the bits in the binary number we can see that the red component of our color is 43, the green 6F, and the blue CD.

These days, hexadecimal notation (often spoken as “hex”) is more common than octal, but as we shall soon see, octal's ability to express three bits of binary will be very useful...

With octal notation we use octal numbers to set the pattern of desired permissions. Since each digit in an octal number represents three binary digits, this maps nicely to the

scheme used to store the file mode. This table shows what we mean:

! 6 - % + 9 *)

Octal	Binary	File Mode
$	$$$	
"	$$"	
*	$"$	8
/	$""	8
&	"$$	
#	"$"	
+	""$	8
0	""""	8

By using three octal digits, we can set the file mode for the owner, group owner, and world:

```
                    &

8 8   "       $ *$"+ $/ $+ "& #*   <
                      N**

8     "       $ *$"+ $/ $+ "& #*   <
```

By passing the argument “600”, we were able to set the permissions of the owner to read and write while removing all permissions from the group owner and world. Though remembering the octal to binary mapping may seem inconvenient, you will usually only have to use a few common ones: 7 (8), 6 (8), 5 ( ), 4 ( ), and 0 ( ).

also supports a symbolic notation for specifying file modes. Symbolic notation is divided into three parts: who the change will affect, which operation will be performed, and what permission will be set. To specify who is affected, a combination of the characters “u”, “g”, “o”, and “a” is used as follows:

! 6 2 # ! >

Symbol	Meaning
	Short for “user” but means the file or directory owner.
	Group owner.

	Short for “others,” but means world.
	Short for “all.” The combination of “u”, “g”, and “o”.

If no character is specified, “all” will be assumed. The operation may be a “+” indicating that a permission is to be added, a “-” indicating that a permission is to be taken away, or a “=” indicating that only the specified permissions are to be applied and that all others are to be removed.

Permissions are specified with the “r”, “w”, and “x” characters. Here are some examples of symbolic notation:

! 6 3 # ! > 1

Notation	Meaning
7	Add execute permission for the owner.
	Remove execute permission from the owner.
7	Add execute permission for the owner, group, and world. Equivalent to 7 .
8	Remove the read and write permission from anyone besides the owner and group owner.
L 8	Set the group owner and anyone besides the owner to have read and write permission. If either the group owner or world previously had execute permissions, they are removed.
7 > L	Add execute permission for the owner and set the permissions for the group and others to read and execute. Multiple specifications may be separated by commas.

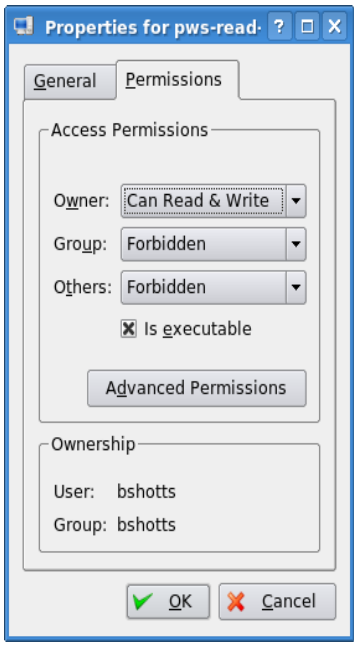
Some people prefer to use octal notation, some folks really like the symbolic. Symbolic notation does offer the advantage of allowing you to set a single attribute without disturbing any of the others.

Take a look at the man page for more details and a list of options. A word of caution regarding the “--recursive” option: it acts on both files and directories, so it's not as useful as one would hope since, we rarely want files and directories to have the same permissions.

Setting File Mode With The GUI

Now that we have seen how the permissions on files and directories are set, we can better

understand the permission dialogs in the GUI. In both Nautilus (GNOME) and Konqueror (KDE), right-clicking a file or directory icon will expose a properties dialog. Here is an example from KDE 3.5:



" .1 ('2
7 .

Here we can see the settings for the owner, group, and world. In KDE, clicking the “Advanced Permissions” button brings up another dialog that allows you to set each of the mode attributes individually. Another victory for understanding brought to us by the command line!

– Set Default Permissions

The command controls the default permissions given to a file when it is created. It uses octal notation to express a of bits to be removed from a file's mode attributes. Let's take a look:

```
$$$*
                    &
```

```
8 8   "       $ *$"+ $/ $+ "& #/   <
```

We first removed any old copy of < to make sure we were starting fresh. Next, we ran the command without an argument to see the current value. It responded with the value $$$* (the value $$** is another common default value), which is the octal representation of our mask. We next create a new instance of the file < and observe its permissions.

We can see that both the owner and group get read and write permission, while everyone else only gets read permission. The reason that world does not have write permission is because of the value of the mask. Let's repeat our example, this time setting the mask ourselves:

```
                    ****
                  &

8 8 8 "       $ *$"+ $/ $+ "& #!   <
```

When we set the mask to $$$$ (effectively turning it off), we see that the file is now world writable. To understand how this works, we have to look at octal numbers again. If we take the mask and expand it into binary, and then compare it to the attributes we can see what happens:

Original file mode	8 8 8
Mask	$$$ $$$ $$$ $"$
Result	8 8

Ignore for the moment the leading zeros (we'll get to those in a minute) and observe that where the 1 appears in our mask, an attribute was removed—in this case, the world write permission. That's what the mask does. Everywhere a 1 appears in the binary value of the mask, an attribute is unset. If we look at a mask value of $$** , we can see what it does:

Original file mode	8 8 8
Mask	$$$ $$$ $"$ $"$
Result	8

Again, where a 1 appears in the binary value, the corresponding attribute is unset. Play with some values (try some sevens) to get used to how this works. When you're done, remember to clean up:

```
                    #    ***
```

Most of the time we won't have to change the mask; the default provided by your distribution will be fine. In some high-security situations, however, we will want to control it.

Some Special Permissions

Though we usually see an octal permission mask expressed as a three digit number, it is more technically correct to express it in four digits. Why? Because, in addition to read, write, and execute permission, there are some other, less used, permission settings.

The first of these is the ! (octal 4000). When applied to an executable file, it sets the +. from that of the real user (the user actually running the program) to that of the program's owner. Most often this is given to a few programs owned by the superuser. When an ordinary user runs a program that is “ ” , the program runs with the effective privileges of the superuser. This allows the program to access files and directories that an ordinary user would normally be prohibited from accessing. Clearly, because this raises security concerns, the number of setuid programs must be held to an absolute minimum.

The second less-used setting is the ! (octal 2000) which, like the setuid bit, changes the +. from the +. of the real user to that of the file owner. If the setgid bit is set on a directory, newly created files in the directory will be given the group ownership of the directory rather the group ownership of the file's creator. This is useful in a shared directory when members of a common group need access to all the files in the directory, regardless of the file owner's primary group.

The third is called the ! (octal 1000). This is a holdover from ancient Unix, where it was possible to mark an executable file as “not swappable.” On files, Linux ignores the sticky bit, but if applied to a directory, it prevents users from deleting or renaming files unless the user is either the owner of the directory, the owner of the file, or the superuser. This is often used to control access to a shared directory, such as 5 6 .

Here are some examples of using with symbolic notation to set these special permissions. First assigning setuid to a program:

7

Next, assigning setgid to a directory:

7

Finally, assigning the sticky bit to a directory:

7

When viewing the output from , you can determine the special permissions. Here are some examples. First, a program that is setuid:

8

A directory that has the setgid attribute:

8 8

A directory with the sticky bit set:

8 8 8

Changing Identities

At various times, we may find it necessary to take on the identity of another user. Often we want to gain superuser privileges to carry out some administrative task, but it is also possible to “become” another regular user for such things as testing an account. There are three ways to take on an alternate identity:

1. Log out and log back in as the alternate user.
2. Use the command.
3. Use the command.

We will skip the first technique since we know how to do it and it lacks the convenience of the other two. From within our own shell session, the command allows you to assume the identity of another user, and either start a new shell session with that user's IDs, or to issue a single command as that user. The command allows an administrator to set up a configuration file called 5 5 , and define specific commands that particular users are permitted to execute under an assumed identity. The choice of which command to use is largely determined by which Linux distribution you use. Your distribution probably includes both commands, but its configuration will favor either one or the other. We'll start with .

– Run A Shell With Substitute User And Group IDs

The command is used to start a shell as another user. The command syntax looks like this:

```
( ( )) (              )
```

If the “-l” option is included, the resulting shell session is a for the specified user. This means that the user's environment is loaded and the working directory is changed to the user's home directory. This is usually what we want. If the user is not specified, the superuser is assumed. Notice that (strangely) the “-l” may be abbreviated “-”, which is how it is most often used. To start a shell for the superuser, we would do this:

```
: 8
        Z
```

After entering the command, we are prompted for the superuser's password. If it is successfully entered, a new shell prompt appears indicating that this shell has superuser privileges (the trailing “#” rather than a “$”) and the current working directory is now the home directory for the superuser (normally 5). Once in the new shell, we can carry out commands as the superuser. When finished, enter “exit” to return to the previous shell:

```
        Z
```

It is also possible to execute a single command rather than starting a new interactive command by using this way:

```
  J          J
```

Using this form, a single command line is passed to the new shell for execution. It is important to enclose the command in quotes, as we do not want expansion to occur in our shell, but rather in the new shell:

```
                                  %          %
:  8
  8      "            0#& *$$0 $! "" $/ "% 5    5          <

5   5-
    $
```

sudo – Execute A Command As Another User

The command is like in many ways, but has some important additional capabilities. The administrator can configure to allow an ordinary user to execute commands as a different user (usually the superuser) in a very controlled way. In particular, a user may be restricted to one or more specific commands and no others. Another important difference is that the use of does not require access to the superuser's password. To authenticate using , the user uses his/her own password. Let's say, for example, that has been configured to allow us to run a fictitious backup program called “backup_script”, which requires superuser privileges. With it would be done like this:

```
                                  .
:  8
(,    G    6 (      <<<
```

After entering the command, we are prompted for our password (not the superuser's) and once the authentication is complete, the specified command is carried out. One important difference between and is that does not start a new shell, nor does it load another user's environment. This means that commands do not need to be quoted any differently than they would be without using . Note that this behavior can be overridden by specifying various options. See the man page for details.

To see what privileges are granted by , use the “-l” option to list them:

```
2       ,          8
   M3==O 3==
```

Ubuntu And

One of the recurrent problems for regular users is how to perform certain tasks that require superuser privileges. These tasks include installing and updating software, editing system configuration files, and accessing devices. In the Windows world, this is often done by giving users administrative privileges. This allows users to perform these tasks. However, it also enables programs executed by the user to have the same abilities. This is desirable in most cases, but it also permits (malicious software) such as viruses to have free reign of the computer.

In the Unix world, there has always been a larger division between regular users and administrators, owing to the multiuser heritage of Unix. The approach taken in Unix is to grant superuser privileges only when needed. To do this, the and commands are commonly used.

Up until a few of years ago, most Linux distributions relied on for this purpose. didn't require the configuration that required, and having a root account is traditional in Unix. This introduced a problem. Users were tempted to operate as root unnecessarily. In fact, some users operated their systems as the root user exclusively, since it does away with all those annoying “permission denied” messages. This is how you reduce the security of a Linux system to that of a Windows system. Not a good idea.

When Ubuntu was introduced, its creators took a different tack. By default, Ubuntu disables logins to the root account (by failing to set a password for the account), and instead uses to grant superuser privileges. The initial user account is granted full access to superuser privileges via and may grant similar powers to subsequent user accounts.

chown – Change File Owner And Group

The 8 command is used to change the owner and group owner of a file or directory. Superuser privileges are required to use this command. The syntax of 8 looks like this:

```
(  )(2(  ))
```

8 can change the file owner and/or the file group owner depending on the first argument of the command. Here are some examples:

! 6 4 * 1

Argument	Results
	Changes the ownership of the file from its current owner to user .
	Changes the ownership of the file from its current owner to user and changes the file group owner to group .
	Changes the group owner to the group . The file owner is unchanged.
	Change the file owner from the current owner to user and changes the group owner to the login group of user .

Let's say that we have two users; , who has access to superuser privileges and , , who does not. User wants to copy a file from her home directory to the home directory of user , . Since user wants , to be able to edit the file, changes the ownership of the copied file from to , :

```
                                    !          !
:  8
                                        !  !
  8      "              *$"+ $/ *$ "& /$ 5    5   ,5 ,    <
                                       !2     !  !
                                        !  !
  8      "    ,     ,    , *$"+ $/ *$ "& /$ 5    5   ,5 ,    <
```

Here we see user copy the file from her directory to the home directory of user , . Next, changes the ownership of the file from (a result of using) to , . Using the trailing colon in the first argument, also changed the group ownership of the file to the login group of , , which happens to be group , .

Notice that after the first use of , was not prompted for her password. This is because , in most configurations, “trusts” you for several minutes until its timer runs out.

chgrp – Change Group Ownership

In older versions of Unix, the 8 command only changed file ownership, not group ownership. For that purpose, a separate command, 6 was used. It works much the

same way as 8 , except for being more limited.

Exercising Our Privileges

Now that we have learned how this permissions thing works, it's time to show it off. We are going to demonstrate the solution to a common problem—setting up a shared directory. Let's imagine that we have two users named “bill” and “karen.” They both have music CD collections and wish to set up a shared directory, where they will each store their music files as Ogg Vorbis or MP3. User has access to superuser privileges via .

The first thing that needs to happen is creating a group that will have both and as members. Using the graphical user management tool, creates a group called and adds users and to it:

(* > ? / ?>)%1

Next, creates the directory for the music files:

O

: 8

Since is manipulating files outside his home directory, superuser privileges are required. After the directory is created, it has the following ownerships and permissions:

```
                                        O
8      *        &$%+ *$"+ $/ *" "! $# 5  5    5    5-
```

As we can see, the directory is owned by and has 755 permissions. To make this directory sharable, needs to change the group ownership and the group permissions to allow writing:

```
                                  2              O
                                  CC"            O
                                        O
8 8    *        &$%+ *$"+ $/ *" "! $# 5  5    5    5-
```

So what does this all mean? It means that we now have a directory, 5 5 5 5- that is owned by and allows read and write access to group . Group has members and , thus and can create files in directory 5 5 5 5- . Other users can list the contents of the directory but cannot create files there.

But we still have a problem. With the current permissions, files and directories created within the Music directory will have the normal permissions of the users and :

```
                        &            O      .
                                        O
8      "          $ *$"+ $/ *& *$ $/    N
```

Actually there are two problems. First, the default on this system is 0022 which prevents group members from writing files belonging to other members of the group. This would not be a problem if the shared directory only contained files, but since this directory will store music, and music is usually organized in a hierarchy of artists and albums, members of the group will need the ability to create files and directories inside directories created by other members. We need to change the used by and to 0002 instead.

Second, each file and directory created by one member will be set to the primary group of the user rather than the group . This can be fixed by setting the setgid bit on the directory:

```
                              5            O
```

```
                                        O
8 8    *        &$%+ *$"+ $/ *& *$ $/ 5  5    5    5-
```

Now we test to see if the new permissions fix the problem. sets his to 0002, removes the previous test file, and creates a new test file and directory:

```
                        ***
                                O      .
                    &           O      .
                                  O      .
                                  O
8 8    *        &$%+ *$"+ $/ *& *$ *&    N
8 8    "          $ *$"+ $/ *& *$ **    N
```

Both files and directories are now created with the correct permissions to allow all members of the group to create files and directories inside the - directory.

The one remaining issue is . The necessary setting only lasts until the end of session and must be reset. In Chapter 11, we'll look at making the change to permanent.

Changing Your Password

The last topic we'll cover in this chapter is setting passwords for yourself (and for other users if you have access to superuser privileges). To set or change a password, the 6 8 command is used. The command syntax looks like this:

```
    (          )
```

To change your password, just enter the 6 8 command. You will be prompted for your old password and your new password:

```
M     O 2K@V 6  8
K 8 2K@V 6  8
```

The 6 8 command will try to enforce use of “strong” passwords. This means it will refuse to accept passwords that are too short, too similar to previous passwords, are dic-

tionary words, or are too easily guessed:

```
M      O 2K@V 6   8
K 8 2K@V 6   8
G39 :3((.SQ9
K 8 2K@V 6   8
G39 :3((.SQ9       .3U
K 8 2K@V 6   8
G39 :3((.SQ9                    , 8
```

If you have superuser privileges, you can specify a username as an argument to the 6 8 command to set the password for another user. Other options are available to the superuser to allow account locking, password expiration, etc. See the 6 8 man page for details.

Summing Up

In this chapter we have seen how Unix-like systems such as Linux manage user permissions to allow the read, write, and execution access to files and directories. The basic ideas of this system of permissions date back to the early days of Unix and have stood up pretty well to the test of time. But the native permissions mechanism in Unix-like systems lacks the fine granularity of more modern systems.

Further Reading

Wikipedia has a good article on malware:
http://en.wikipedia.org/wiki/Malware

There are number of command line programs used to create and maintain users and groups. For more information, see the man pages for the following commands:

6

0 2

Modern operating systems are usually , meaning that they create the illusion of doing more than one thing at once by rapidly switching from one executing program to another. The Linux kernel manages this through the use of . Processes are how Linux organizes the different programs waiting for their turn at the CPU.

Sometimes a computer will become sluggish or an application will stop responding. In this chapter, we will look at some of the tools available at the command line that let us examine what programs are doing, and how to terminate processes that are misbehaving.

This chapter will introduce the following commands:

6 – Report a snapshot of current processes

6 – Display tasks

– List active jobs

– Place a job in the background

– Place a job in the foreground

– Send a signal to a process

– Kill processes by name

8 – Shutdown or reboot the system

How A Process Works

When a system starts up, the kernel initiates a few of its own activities as processes and launches a program called . , in turn, runs a series of shell scripts (located in 5) called , which start all the system services. Many of these services are implemented as , programs that just sit in the background and do their thing without having any user interface. So even if we are not logged in, the system is at least a little busy performing routine stuff.

The fact that a program can launch other programs is expressed in the process scheme as a producing a .

The kernel maintains information about each process to help keep things organized. For example, each process is assigned a number called a +. or 7+. . PIDs are assigned in ascending order, with always getting PID 1. The kernel also keeps track of the memory assigned to each process, as well as the processes' readiness to resume execution. Like files, processes also have owners and user IDs, effective user IDs, etc.

Viewing Processes

The most commonly used command to view processes (there are several) is 6 . The 6 program has a lot of options, but in its simplest form it is used like this:

```
 :@9 ))U        )@-'  -9
 #"%! 6  5"    $$ $$ $$
"$"*% 6  5"    $$ $$ $$ 6
```

The result in this example lists two processes, process 5198 and process 10129, which are and 6 respectively. As we can see, by default, 6 doesn't show us very much, just the processes associated with the current terminal session. To see more, we need to add some options, but before we do that, let's look at the other fields produced by 6 .))U is short for “Teletype,” and refers to the for the process. Unix is showing its age here. The)@-' field is the amount of CPU time consumed by the process. As we can see, neither process makes the computer work very hard.

If we add an option, we can get a bigger picture of what the system is doing:

```
 :@9 ))U     ()3)   )@-'  S--3K9
 *0%% E       (     $ $$ 5   5      5                 W
 *!*$ E       (     $ $" 5   5      5              "<"$
"#+&0 E       (     $ $$ 5   5   5   5   5
"#0#" E       (     $ $$ 5   5   5        5   5   5
"#0#& E       (     $ $$ 5   5   5             8
"#0## E       (     $ $" 5   5              6    6   & W6
"#00& E       (     $ $* 5   5   5 6        W
"#0%/ E       (     $ $$     N         8      6 7      N
"#0%& E       (     $ $$       Q     <<<
"#0%0 E       (     $ $$    6      W


    ,    <<<
```

Adding the “x” option (note that there is no leading dash) tells 6 to show all of our processes regardless of what terminal (if any) they are controlled by. The presence of a “?” in the))U column indicates no controlling terminal. Using this option, we see a list of every process that we own.

Since the system is running a lot of processes, 6 produces a long list. It is often helpful to pipe the output from 6 into for easier viewing. Some option combinations also produce long lines of output, so maximizing the terminal emulator window may be a good idea, too.

A new column titled ()3) has been added to the output. ()3) is short for “state” and reveals the current status of the process:

! @ 7 #

State	Meaning
Q	Running. This means that the process is running or ready to run.
(	Sleeping. The process is not running; rather, it is waiting for an event, such as a keystroke or network packet.
9	Uninterruptible Sleep. Process is waiting for I/O such as a disk drive.
)	Stopped. Process has been instructed to stop. More on this later.
H	A defunct or “zombie” process. This is a child process that has terminated, but has not been cleaned up by its parent.
X	A high priority process. It's possible to grant more importance to a process, giving it more time on the CPU. This property of a process is called . A process with high priority is said to be less because it's taking more of the CPU's time, which leaves less for everybody else.
K	A low priority process. A process with low priority (a “nice” process) will only get processor time after other processes with higher priority have been serviced.

The process state may be followed by other characters. These indicate various exotic process characteristics. See the 6 man page for more detail.

Another popular set of options is “aux” (without a leading dash). This gives us even more information:

```
2('Q   :@9 4 :2 4-'-   ;(H   Q(( ))U     ()3) ()3Q)   )@-'  S--3K9
       "  $<$  $<$   *"'/+   +&& E       (    -  $#   $ /"
       *  $<$  $<$     $    $ E       (X   -  $#   $ $$
       /  $<$  $<$     $    $ E       (X   -  $#   $ $$
       &  $<$  $<$     $    $ E       (X   -  $#   $ $$
       #  $<$  $<$     $    $ E       (X   -  $#   $ $+  8
       +  $<$  $<$     $    $ E       (X   -  $#   $ /+
       0  $<$  $<$     $    $ E       (X   -  $#   $ $$



    ,    <<<
```

This set of options displays the processes belonging to every user. Using the options without the leading dash invokes the command with “BSD style” behavior. The Linux version of 6 can emulate the behavior of the 6 program found in several different Unix implementations. With these options, we get these additional columns:

! @ " 9#. # ,

Header	Meaning
2('Q	User ID. This is the owner of the process.
4 :2	CPU usage in percent.
4-'-	Memory usage in percent.
;(H	Virtual memory size.
Q((	Resident Set Size. The amount of physical memory (RAM) the process is using in kilobytes.
()3Q)	Time when the process started. For values over 24 hours, a date is used.

Viewing Processes Dynamically With 6

While the 6 command can reveal a lot about what the machine is doing, it provides only a snapshot of the machine's state at the moment the 6 command is executed. To see a more dynamic view of the machine's activity, we use the 6 command:

```

```

The 6 program displays a continuously updating (by default, every 3 seconds) display of the system processes listed in order of process activity. The name “top” comes from the fact that the 6 program is used to see the “top” processes on the system. The 6 display consists of two parts: a system summary at the top of the display, followed by a table of processes sorted by CPU activity:

```
 6  "& #% *$  6  + /$>  *     >            $<$0> $<$*> $<$$
)    "$%     >  "       > "$+    6  >  $   66  >  * B
 6 M O  $<04  >  "<$4 ,>  $<$4  > %!</4  >  $<$48 >  $<$4  >  $<$4
-     /"%&%+      >  /"&!+$     >    &+/+     >   "%/%*
(8 6   !0##$$     >  "&%"*!     >   0*+/0*     >  ""&+0+

 :@9 2('Q     :Q  K@  ;@Q)  Q'(  (RQ ( 4 :2 4-'-    )@-'7   S--3K9
 +*&&          /%  "% /"0#* /"*& *"!! (  +</  "<$  "+ *&<&*
""$0"          *$   $  */$& "$%*  !&$ Q  "</  $</   $ $$<"&   6
 +"!$          *$   $  *0$$ ""$$  00* (  $<0  $</   $ $/<++
 +/*"          *$   $ *$%&& 0*&! +#+$ (  $<0  *</   * #"</!
 &%##           *$   $  "$&  %++! #00+ (  $</  /<$   * "%</% V
   "          *$   $  *%0+  #*!  &0+ (  $<$  $<*   $ $/<"&
   *          "#   #     $    $    $ (  $<$  $<$   $ $$<$$
   /          Q)   #     $    $    $ (  $<$  $<$   $ $$<$$
   &           "#   #     $    $    $ (  $<$  $<$   $ $$<0*       A
   #           Q)   #     $    $    $ (  $<$  $<$   $ $$<$& 8
   +           "#   #     $    $    $ (  $<$  $<$   $ $$<&*      5$
   0           "#   #     $    $    $ (  $<$  $<$   $ $$<$+     6
  &"           "#   #     $    $    $ (  $<$  $<$   $ $"<$!       5
  +0           "#   #     $    $    $ (  $<$  $<$   $ $$<$$
 ""&           *$   $     $    $    $ (  $<$  $<$   $ $"<+* 6
 ""+           "#   #     $    $    $ (  $<$  $<$   $ $*<&&   8 6 $
```

The system summary contains a lot of good stuff. Here's a rundown:

! @ (+

Row	Field	Meaning
1	6	Name of the program.
	"& #% *$	Current time of day.
	6 + /$	This is called . It is the amount of time since the machine was last booted. In this example, the system has been up for six and a half hours.
	*	There are two users logged in.
		refers to the number of processes

		that are waiting to run, that is, the number of processes that are in a runnable state and are sharing the CPU. Three values are shown, each for a different period of time. The first is the average for the last 60 seconds, the next the previous 5 minutes, and finally the previous 15 minutes. Values under 1.0 indicate that the machine is not busy.
2	)	This summarizes the number of processes and their various process states.
3	6 M O	This row describes the character of the activities that the CPU is performing.
	$<04	0.7% of the CPU is being used for . This means processes outside of the kernel itself.
	"<$4 ,	1.0% of the CPU is being used for (kernel) processes.
	$<$4	0.0% of the CPU is being used by “nice” (low priority) processes.
	%!</4	98.3% of the CPU is idle.
	$<$48	0.0% of the CPU is waiting for I/O.
4	-	Shows how physical RAM is being used.
5	(8 6	Shows how swap space (virtual memory) is being used.

The 6 program accepts a number of keyboard commands. The two most interesting are , which displays the program's help screen, and A, which quits 6 .

Both major desktop environments provide graphical applications that display information similar to 6 (in much the same way that Task Manager in Windows works), but 6 is better than the graphical versions because it is faster and it consumes far fewer system resources. After all, our system monitor program shouldn't be the source of the system slowdown that we are trying to track.

Controlling Processes

Now that we can see and monitor processes, let's gain some control over them. For our

experiments, we're going to use a little program called as our guinea pig. The program is a sample program supplied with the X Window System (the underlying engine that makes the graphics on our display go) which simply displays a re-sizable window containing the X logo. First, we'll get to know our test subject:

After entering the command, a small window containing the logo should appear somewhere on the screen. On some systems, may print a warning message, but it may be safely ignored.

(6 If your system does not include the program, try using or 8 instead.

We can verify that is running by resizing its window. If the logo is redrawn in the new size, the program is running.

Notice how our shell prompt has not returned? This is because the shell is waiting for the program to finish, just like all the other programs we have used so far. If we close the window, the prompt returns.

Interrupting A Process

Let's observe what happens when we run again. First, enter the command and verify that the program is running. Next, return to the terminal window and press .

In a terminal, pressing , a program. This means that we politely asked the program to terminate. After we pressed , the window closed and the shell prompt returned.

Many (but not all) command-line programs can be interrupted by using this technique.

Putting A Process In The Background

Let's say we wanted to get the shell prompt back without terminating the pro-

gram. We can do this by placing the program in the ! . Think of the terminal as having a (with stuff visible on the surface like the shell prompt) and a background (with stuff hidden behind the surface). To launch a program so that it is immediately placed in the background, we follow the command with an “&” character:

```
                    '
"  *!*/+
```

After entering the command, the window appeared and the shell prompt returned, but some funny numbers were printed too. This message is part of a shell feature called A ! . With this message, the shell is telling us that we have started job number 1 (“ " ”) and that it has PID 28236. If we run 6 , we can see our process:

```
 :@9 ))U        )@-'  -9
"$+$/ 6  5"   $$ $$ $$
*!*/+ 6  5"   $$ $$ $$
*!*/% 6  5"   $$ $$ $$ 6
```

The shell's job control facility also gives us a way to list the jobs that have been launched from our terminal. Using the command, we can see this list:

```
" 7  Q                 Y
```

The results show that we have one job, numbered “1”, that it is running, and that the command was Y .

Returning A Process To The Foreground

A process in the background is immune from keyboard input, including any attempt interrupt it with a . To return a process to the foreground, use the command, this way:

```
" 7  Q                 Y
                       8
```

```

```

The command followed by a percent sign and the job number (called a A !) does the trick. If we only have one background job, the jobspec is optional. To terminate , press .

Stopping (Pausing) A Process

Sometimes we'll want to stop a process without terminating it. This is often done to allow a foreground process to be moved to the background. To stop a foreground process, press B . Let's try it. At the command prompt, type , the ' key, then B:

```
" 7 ( 66
```

After stopping , we can verify that the program has stopped by attempting to resize the window. We will see that it appears quite dead. We can either restore the program to the foreground, using the command, or move the program to the background with the command:

```
                    8
" 7    Y
```

As with the command, the jobspec is optional if there is only one job.

Moving a process from the foreground to the background is handy if we launch a graphical program from the command, but forget to place it in the background by appending the trailing “&”.

Why would we want to launch a graphical program from the command line? There are two reasons. First, the program you wish to run might not be listed on the window manager's menus (such as). Secondly, by launching a program from the command line, you might be able to see error messages that would otherwise be invisible if the program were launched graphically. Sometimes, a program will fail to start up when launched from the graphical menu. By launching it from the command line instead, we may see an error message that will reveal the problem. Also, some graphical programs have many interesting and useful command line options.

Signals

The command is used to “kill” processes. This allows us to terminate programs that need killing. Here's an example:

```
                Y
"  *!&$"
                *!&$"
" 7  )
```

We first launch in the background. The shell prints the jobspec and the PID of the background process. Next, we use the command and specify the PID of the process we want to terminate. We could have also specified the process using a jobspec (for example, “%1”) instead of a PID.

While this is all very straightforward, there is more to it than that. The command doesn't exactly “kill” processes, rather it sends them . Signals are one of several ways that the operating system communicates with programs. We have already seen signals in action with the use of and B . When the terminal receives one of these keystrokes, it sends a signal to the program in the foreground. In the case of , a signal called @K)(Interrupt) is sent; with B , a signal called)(): (Terminal Stop). Programs, in turn, “listen” for signals and may act upon them as they are received. The fact that a program can listen and act upon signals allows a program to do things like save work in progress when it is sent a termination signal.

Sending Signals To Processes With

The command is used to send signals to programs. Its most common syntax looks like this:

```
   (                ) BP1
```

If no signal is specified on the command line, then the)'Q- (Terminate) signal is sent by default. The command is most often used to send the following signals:

! @ - #

Number	Name	Meaning
"	R2:	Hangup. This is a vestige of the good old days when terminals were attached to remote

		computers with phone lines and modems. The signal is used to indicate to programs that the controlling terminal has “hung up.” The effect of this signal can be demonstrated by closing a terminal session. The foreground program running on the terminal will be sent the signal and will terminate. This signal is also used by many daemon programs to cause a reinitialization. This means that when a daemon is sent this signal, it will restart and re-read its configuration file. The Apache web server is an example of a daemon that uses the R2: signal in this way.
*	@K)	Interrupt. Performs the same function as the key sent from the terminal. It will usually terminate a program.
%	1@==	Kill. This signal is special. Whereas programs may choose to handle signals sent to them in different ways, including ignoring them all together, the 1@==signal is never actually sent to the target program. Rather, the kernel immediately terminates the process. When a process is terminated in this manner, it is given no opportunity to “clean up” after itself or save its work. For this reason, the 1@==signal should only be used as a last resort when other termination signals fail.
"#	)'Q-	Terminate. This is the default signal sent by the command. If a program is still “alive” enough to receive signals, it will terminate.
"!	SK)	Continue. This will restore a process after a ()S: signal.
"%	()S:	Stop. This signal causes a process to pause without terminating. Like the 1@==signal, it is not sent to the target process, and thus it cannot be ignored.

Let's try out the command:

```
                    '
"  "/#&+
                       "AN
" 7  R   6
```

In this example, we start the program in the background and then send it a R2: signal with . The program terminates and the shell indicates that the background process has received a hangup signal. We may need to press the enter key a couple of times before the message appears. Note that signals may be specified either by number or by name, including the name prefixed with the letters “SIG”:

```
                    '
"  "/+$"
                    P?+   N*
" 7  @     6
                    '
"  "/+$!
                    FPQP?+   N*M
" 7  @     6
```

Repeat the example above and try out the other signals. Remember, we can also use jobspecs in place of PIDs.

Processes, like files, have owners, and you must be the owner of a process (or the superuser) in order to send it signals with .

In addition to the list of signals above, which are most often used with , there are other signals frequently used by the system. Here is a list of other common signals:

! @ 2) #

Number	Name	Meaning
/	[2@)	Quit.
""	('?;	Segmentation Violation. This signal is sent if a program makes illegal use of memory, that is, it tried to write somewhere it was not allowed to.
*$	)():	Terminal Stop. This is the signal sent by the terminal when the B key is pressed. Unlike the ()S: signal, the)(): signal is received by

		the program but the program may choose to ignore it.
*!	.@K R	Window Change. This is the signal sent by the system when a window changes size. Some programs , like 6 and will respond to this signal by redrawing themselves to fit the new window dimensions.

For the curious, a complete list of signals can be seen with the following command:

```

```

Sending Signals To Multiple Processes With

It's also possible to send signals to multiple processes matching a specified program or username by using the command. Here is the syntax:

```
    (              ) (          )
```

To demonstrate, we will start a couple of instances of the program and then terminate them:

```
                    '
"  "!!$"
                    '
*  "!!$*

"   )
*7  )
```

Remember, as with , you must have superuser privileges to send signals to processes that do not belong to you.

Shutting Down The System

The process of shutting down the system involves the orderly termination of all the processes on the system, as well as performing some vital housekeeping chores (like syncing

all of the mounted file systems) before the system powers off. There are four commands that can perform this function. They are , 6 8 , , and 8 . The first three are pretty self-explanatory and are generally used without any command line options. For example:

The 8 command is a bit more interesting. With it, we can specify which of the actions to perform (halt, power down, or reboot), and provide a time delay to the shut-down event. Most often it is used like this:

to halt the system, or like this:

to reboot the system. The delay can be specified in a variety of ways. See the 8 man page for details. Once the 8 command is executed, a message is “broad-cast” to all logged-in users warning them of the impending event.

More Process Related Commands

Since monitoring processes is an important system administration task, there are a lot of commands for it. Here are some to play with:

! @ 3) 7 :

Command	Description
6	Outputs a process list arranged in a tree-like pattern showing the parent/child relationships between processes.
	Outputs a snapshot of system resource usage including, memory, swap and disk I/O. To see a continuous display, follow the command with a time delay (in seconds) for updates. For example: # . Terminate the output with .
	A graphical program that draws a graph showing system load over time.
	Similar to the program, but draws the graph in the terminal. Terminate the output with .

Summing Up

Most modern systems feature a mechanism for managing multiple processes. Linux provides a rich set of tools for this purpose. Given that Linux is the world's most deployed server operating system, this makes a lot of sense. However, unlike some other systems, Linux relies primarily on command line tools for process management. Though there are graphical process tools for Linux, the command line tools are greatly preferred because of their speed and light footprint. While the GUI tools may look pretty, they often create a lot of system load themselves, which somewhat defeats the purpose.

Part 2 – Configuration And The Environment

! $

As we discussed earlier, the shell maintains a body of information during our shell session called the . Data stored in the environment is used by programs to determine facts about the system's configuration. While most programs use
to store program settings, some programs will also look for values stored in the environment to adjust their behavior. Knowing this, we can use the environment to customize our shell experience.

In this chapter, we will work with the following commands:

6 – Print part or all of the environment

– Set shell options

6 – Export environment to subsequently executed programs

– Create an alias for a command

What Is Stored In The Environment?

The shell stores two basic types of data in the environment, though, with > the types are largely indistinguishable. They are ! and ! . Shell variables are bits of data placed there by , and environment variables are everything else. In addition to variables, the shell also stores some programmatic data, namely and . We covered aliases in Chapter 5, and shell functions (which are related to shell scripting) will be covered in Part 4.

Examining The Environment

To see what is stored in the environment, we can use either the builtin in or the 6 program. The command will show both the shell and environment variables, while 6 will only display the latter. Since the list of environment contents will be fairly long, it is best to pipe the output of either command into :

```
0
```

Doing so, we should get something that looks like this:

```
19'N-2=)@R'39L
((RN3?'K)N:@9L++++
RS()K3-'L
?:?N3?'K)N@K SL5  65 6  : S 0 5(< 6       ++!% "
(R'==L5   5
)'Q-L
V9?N-'K2N:Q' @VL
R@()(@H'L"$$$
V9?N('((@SKN SS1@'L+ 0 $# +#!&+ /   /"$" $$&+  * $$
"*$!#*'"%%$<%%+0$# ""00$#+"%%
?)1*NQ N @='(L5   5    *<$5      5    5  5<
*<$ 5    5  5<   5     5      5      *<$
?)1NQ N @='(L5   5   5     5    5  5<     5    5  5<   5    5
 5
?(N=@GL5    5  5<
.@K9S.@9L*%/+$"/+
[)9@QL5   5   5A  /</
[)@K L5   5   5A  /</5
19'N 2==N('((@SKL
2('QL
=(N S=SQ(L  L$$   L$$   L$$T/&   L$$T/+ 6 L&$T//   L$$T/#   L&$T//T$"
   L&$T//T$"   L$"T$#T/0T&"   L$"T$#T/0T&"   L$$T/* D<   L$$T/* D<
```

What we see is a list of environment variables and their values. For example, we see a variable called 2('Q , which contains the value “me”. The 6 command can also list the value of a specific variable:

```
EFGH
```

The command, when used without options or arguments, will display both the shell and environment variables, as well as any defined shell functions. Unlike 6 , its output is courteously sorted in alphabetical order:

```
0
```

It is also possible to view the contents of a variable using the command, like this:

```
                        3RSOG
5   5
```

One element of the environment that neither nor 6 displays is aliases. To see them, enter the command without arguments:

```
    <LJ    <D      L  ,J
     LJ         L  ,J
     LJ       L  ,J
     LJ  J
    8   LJ    P 5   5   58       ,    ,              8
      8    J
```

Some Interesting Variables

The environment contains quite a few variables, and though the environment will differ from the one presented here, we will likely see the following variables in our environment:

! 1 B !

Variable	Contents
9@(:=3U	The name of your display if you are running a graphical environment. Usually this is “:0”, meaning the first display generated by the X server.
'9@)SQ	The name of the program to be used for text editing.
(R'==	The name of your shell program.
RS-'	The pathname of your home directory.
=3K?	Defines the character set and collation order of your language.
S=9N:.9	The previous working directory.
:3?'Q	The name of the program to be used for paging output. This is often set to 5 5 5 .
:3)R	A colon-separated list of directories that are searched when you enter the name of a executable program.
:("	Prompt String 1. This defines the contents of the shell prompt. As we will later see, this can be extensively customized.

:.9	The current working directory.
)'Q-	The name of your terminal type. Unix-like systems support many terminal protocols; this variable sets the protocol to be used with your terminal emulator.
)H	Specifies your timezone. Most Unix-like systems maintain the computer's internal clock in 0 (UTC) and then displays the local time by applying an offset specified by this variable.
2('Q	Your username.

Don't worry if some of these values are missing. They vary by distribution.

How Is The Environment Established?

When we log on to the system, the program starts, and reads a series of configuration scripts called , which define the default environment shared by all users. This is followed by more startup files in our home directory that define our personal environment. The exact sequence depends on the type of shell session being started. There are two kinds: a login shell session and a non-login shell session.

A login shell session is one in which we are prompted for our username and password; when we start a virtual console session, for example. A non-login shell session typically occurs when we launch a terminal session in the GUI.

Login shells read one or more startup files as shown in Table 11-2:

! " # # #

File	Contents
5 56	A global configuration script that applies to all users.
5 < N6	A user's personal startup file. Can be used to extend or override settings in the global configuration script.
5 < N	If 5< N6 is not found, attempts to read this script.
5 <6	If neither 5< N6 nor 5< N is found, attempts to read this file. This is the default in Debian-based distributions, such as Ubuntu.

Non-login shell sessions read the following startup files:

! (# > # #

File	Contents
5 5 <	A global configuration script that applies to all users.
5 <	A user's personal startup file. Can be used to extend or override settings in the global configuration script.

In addition to reading the startup files above, non-login shells also inherit the environment from their parent process, usually a login shell.

Take a look and see which of these startup files are installed. Remember—since most of the filenames listed above start with a period (meaning that they are hidden), we will need to use the "-a" option when using .

The 5< file is probably the most important startup file from the ordinary user's point of view, since it is almost always read. Non-login shells read it by default and most startup files for login shells are written in such a way as to read the 5< file as well.

What's In A Startup File?

If we take a look inside a typical < N6 (taken from a CentOS 6 system), it looks something like this:

```
Z <  N6

Z ?
     5<     T
    < 5<


Z 2   6                6 6

:3)RL :3)R  RS-'5
 6   :3)R
```

Lines that begin with a "#" are and are not read by the shell. These are there for human readability. The first interesting thing occurs on the fourth line, with the following code:

```
     5<     T
```

```
    < 5<
```

This is called an , which we will cover fully when we get to shell scripting in Part 4, but for now, here is a translation:

```
@       _ 5<    _     >
         _ 5<    _    <
```

We can see that this bit of code is how a login shell gets the contents of < . The next thing in our startup file has to do with the :3)R variable.

Ever wonder how the shell knows where to find commands when we enter them on the command line? For example, when we enter , the shell does not search the entire computer to find 5 5 (the full pathname of the command), rather, it searches a list of directories that are contained in the :3)R variable.

The :3)R variable is often (but not always, depending on the distribution) set by the 5 56 startup file and with this code:

```
:3)RL :3)R  RS-'5
```

:3)R is modified to add the directory RS-'5 to the end of the list. This is an example of parameter expansion, which we touched on in Chapter 7. To demonstrate how this works, try the following:

```
                    $J+       J
                     3
)
                    $3  J   J
                     3
)         <
```

Using this technique, we can append text to the end of a variable's contents.

By adding the string RS-'5 to the end of the :3)R variable's contents, the directory RS-'5 is added to the list of directories searched when a command is entered. This means that when we want to create a directory within our home directory for storing our own private programs, the shell is ready to accommodate us. All we have to do is call

it , and we're ready to go.

2 : Many distributions provide this :3)R setting by default. Debian based distributions, such as Ubuntu, test for the existence of the 5 directory at login, and dynamically add it to the :3)R variable if the directory is found.

Lastly, we have:

```
6 :3)R
```

The 6 command tells the shell to make the contents of :3)R available to child processes of this shell.

Modifying The Environment

Since we know where the startup files are and what they contain, we can modify them to customize our environment.

Which Files Should We Modify?

As a general rule, to add directories to your :3)R , or define additional environment variables, place those changes in < N6 (or equivalent, according to your distribution. For example, Ubuntu uses <6). For everything else, place the changes in < .

2 6 Unless you are the system administrator and need to change the defaults for all users of the system, restrict your modifications to the files in your home directory. It is certainly possible to change the files in 5 such as 6 , and in many cases it would be sensible to do so, but for now, let's play it safe.

Text Editors

To edit (i.e., modify) the shell's startup files, as well as most of the other configuration files on the system, we use a program called a . A text editor is a program that is, in some ways, like a word processor in that it allows us to edit the words on the screen with a moving cursor. It differs from a word processor by only supporting pure text, and often contains features designed for writing programs. Text editors are the central tool used by software developers to write code, and by system administrators to manage the

configuration files that control the system.

There are a lot of different text editors available for Linux; most systems have several installed. Why so many different ones? Because programmers like writing them, and since programmers use them extensively, they write editors to express their own desires as to how they should work.

Text editors fall into two basic categories: graphical and text based. GNOME and KDE both include some popular graphical editors. GNOME ships with an editor called , which is usually called “Text Editor” in the GNOME menu. KDE usually ships with three which are (in order of increasing complexity) , 8 , and .

There are many text-based editors. The popular ones we'll encounter are , , and . The editor is a simple, easy-to-use editor designed as a replacement for the 6 editor supplied with the PINE email suite. The editor (on most Linux systems replaced by a program named , which is short for “Vi IMproved”) is the traditional editor for Unix-like systems. It will be the subject of our next chapter. The editor was originally written by Richard Stallman. It is a gigantic, all-purpose, does-everything programming environment. While readily available, it is seldom installed on most Linux systems by default.

Using A Text Editor

Text editors can be invoked from the command line by typing the name of the editor followed by the name of the file you want to edit. If the file does not already exist, the editor will assume that we want to create a new file. Here is an example using :

```
.
```

This command will start the text editor and load the file named “some_file”, if it exists.

Graphical text editors are pretty self-explanatory, so we won't cover them here. Instead, we will concentrate on our first text-based text editor, . Let's fire up and edit the < file. But before we do that, let's practice some “safe computing.” Whenever we edit an important configuration file, it is always a good idea to create a backup copy of the file first. This protects us in case we mess the file up while editing. To create a backup of the < file, do this:

```
6 <  <  <
```

It doesn't matter what we call the backup file, just pick an understandable name. The extensions “.bak”, “.sav”, “.old”, and “.orig” are all popular ways of indicating a backup file. Oh, and remember that 6 will silently.

Now that we have a backup file, we'll start the editor:

Once starts, we’ll get a screen like this:

2 6 If your system does not have installed, you may use a graphical editor instead.

The screen consists of a header at the top, the text of the file being edited in the middle and a menu of commands at the bottom. Since was designed to replace the text editor supplied with an email client, it is rather short on editing features.

The first command you should learn in any text editor is how to exit the program. In the case of , you type to exit. This is indicated in the menu at the bottom of

the screen. The notation “^X” means . This is a common notation for control characters used by many programs.

The second command we need to know is how to save our work. With it's . With this knowledge under our belts, we're ready to do some editing. Using the down arrow key and/or the : 9 8 key, move the cursor to the end of the file, then add the following lines to the < file:

```
***
 RPF+=S?+HST$
 RPF+FP@G$ ***
  $%          $   %
  $%        $   %
```

2 6 Your distribution may already include some of these, but duplicates won't hurt anything.

Here is the meaning of our additions:

! - * '!

Line	Meaning
$$$*	Sets the to solve the problem with shared directories we discussed in Chapter 9.
6 R@() SK)QS=L 6	Causes the shell's history recording feature to ignore a command if the same command was just recorded.
6 R@()(@H'L"$$$	Increases the size of the command history from the usual default of 500 lines to 1000 lines.
<LJ <D L J	Creates a new command called “ < ” which displays all directory entries that begin with a dot.
LJ L J	Creates a new command called “ ” which displays a long format directory listing.

As we can see, many of our additions are not intuitively obvious, so it would be a good idea to add some comments to our < file to help explain things to the humans. Using the editor, change our additions to look like this:

```
U =                    !
    $$$*

U P                          !
U     !  -     ***
 6   R@() SK)QS=L       6
 6   R@()(@H'L"$$$

U :
     <LJ     <D       L    J
      LJ           L   J
```

Ah, much better! With our changes complete, press to save our modified < file, and to exit .

Why Comments Are Important

Whenever you modify configuration files it's a good idea to add some comments to document your changes. Sure, you'll probably remember what you changed tomorrow, but what about six months from now? Do yourself a favor and add some comments. While you're at it, it’s not a bad idea to keep a log of what changes you make.

Shell scripts and startup files use a “#” symbol to begin a comment. Other configuration files may use other symbols. Most configuration files will have comments. Use them as a guide.

You will often see lines in configuration files that are to prevent them from being used by the affected program. This is done to give the reader suggestions for possible configuration choices or examples of correct configuration syntax. For example, the < file of Ubuntu 14.04 contains these lines:

```
Z
Z       LJ     J
Z       LJ    3J
Z      LJ     J
```

The last three lines are valid alias definitions that have been commented out. If you remove the leading "#" symbols from these three lines, a technique called , you will activate the aliases. Conversely, if you add a "#" symbol to the beginning of a line, you can deactivate a configuration line while preserving the information it contains.

Activating Our Changes

The changes we have made to our < will not take affect until we close our terminal session and start a new one, since the < file is only read at the beginning of a session. However, we can force to re-read the modified < file with the following command:

After doing this, we should be able to see the effect of our changes. Try out one of the new aliases:

Summing Up

In this chapter we learned an essential skill—editing configuration files with a text editor. Moving forward, as we read man pages for commands, take note of the environment variables that commands support. There may be a gem or two. In later chapters, we will learn about shell functions, a powerful feature that you can also include in the startup files to add to your arsenal of custom commands.

Further Reading

The INVOCATION section of the man page covers the startup files in gory detail.

(3

There is an old joke about a visitor to New York City asking a passerby for directions to the city's famous classical music venue:

Visitor: Excuse me, how do I get to Carnegie Hall?

Passerby: Practice, practice, practice!

Learning the Linux command line, like becoming an accomplished pianist, is not something that we pick up in an afternoon. It takes years of practice. In this chapter, we will introduce the (pronounced “vee eye”) text editor, one of the core programs in the Unix tradition. is somewhat notorious for its difficult user interface, but when we see a master sit down at the keyboard and begin to “play,” we will indeed be witness to some great art. We won't become masters in this chapter, but when we are done, we will know how to play “chopsticks” in .

Why We Should Learn vi

In this modern age of graphical editors and easy-to-use text-based editors such as , why should we learn ? There are three good reasons:

> is always available. This can be a lifesaver if we have a system with no graphical interface, such as a remote server or a local system with a broken X configuration. , while increasingly popular, is still not universal. POSIX, a standard for program compatibility on Unix systems, requires that be present.
>
> is lightweight and fast. For many tasks, it's easier to bring up than it is to find the graphical text editor in the menus and wait for its multiple megabytes to load. In addition, is designed for typing speed. As we shall see, a skilled user never has to lift his or her fingers from the keyboard while editing.
>
> We don't want other Linux and Unix users to think we are sissies.

Okay, maybe two good reasons.

A Little Background

The first version of was written in 1976 by Bill Joy, a University of California at Berkley student who later went on to co-found Sun Microsystems. derives its name from the word “visual,” because it was intended to allow editing on a video terminal with a moving cursor. Previous to , there were which operated on a single line of text at a time. To specify a change, we tell a line editor to go to a particular line and describe what change to make, such as adding or deleting text. With the advent of video terminals (rather than printer-based terminals like teletypes) visual editing became possible. actually incorporates a powerful line editor called , and we can use line editing commands while using .

Most Linux distributions don't include real ; rather, they ship with an enhanced replacement called (which is short for “vi improved”) written by Bram Moolenaar. is a substantial improvement over traditional Unix and is usually symbolically linked (or aliased) to the name “vi” on Linux systems. In the discussions that follow, we will assume that we have a program called “vi” that is really .

Starting And Stopping vi

To start , we simply enter the following:

```
```

And a screen like this should appear:

```
                ;@-   ;  @ 6

                      0<"<"/!
              , G    -          <
        ;      6              ,

             (6     ;       6    F
       ,6      6  6    X'    C

       ,6    AX'    C
       ,6      6X'    C     X "C              6
       ,6      6       0X'    C

            Q        ;    6
       ,6         6X'    C          ;
```

```
      ,6     6 6      X'   C
```

Just as we did with earlier, the first thing to learn is how to exit. To exit, we enter the following command (note that the colon character is part of the command):

```
2,
```

The shell prompt should return. If, for some reason, will not quit (usually because we made a change to a file that has not yet been saved), we can tell that we really mean it by adding an exclamation point to the command:

```
2,L
```

(6 If you get “lost” in , try pressing the ' key twice to find your way again.

Compatibility Mode

In the example startup screen above, we see the text “Running in Vi compatible mode.” This means that will run in a mode that is closer to the normal behavior of rather than the enhanced behavior of . For purposes of this chapter, we will want to run with its enhanced behavior. To do this, you have a few options: Try running instead of . If that works, consider adding LJ J to your < file. Alternatively, use this command to add a line to your configuration file:

```
_      6_ CC  5<
```

Different Linux distributions package in different ways. Some distributions install a minimal version of by default that only supports a limited set of features. While preforming the lessons that follow, you may encounter missing features. If this is the case, install the full version of .

Editing Modes

Let's start up again, this time passing to it the name of a nonexistent file. This is how we can create a new file with :

If all goes well, we should get a screen like this:

The leading tilde characters (”~”) indicate that no text exists on that line. This shows that we have an empty file. / (3

The second most important thing to learn about (after learning how to exit) is that is a . When starts up, it begins in . In this mode, almost every key is a command, so if we were to start typing, would basically go crazy and make a big mess.

Entering Insert Mode

In order to add some text to our file, we must first enter . To do this, we press the “i” key. Afterward, we should see the following at the bottom of the screen if is running in its usual enhanced mode (this will not appear in compatible mode):

```
P?FGH+
```

Now we can enter some text. Try this:

```
+ ,                    -!
```

To exit insert mode and return to command mode, press the ' key.

Saving Our Work

To save the change we just made to our file, we must enter an while in command mode. This is easily done by pressing the “:” key. After doing this, a colon character should appear at the bottom of the screen:

```

```

To write our modified file, we follow the colon with a “w” then Enter:

```

```

The file will be written to the hard drive and we should get a confirmation message at the bottom of the screen, like this:

```
_ < _ K 8 "=> &+ 8
```

(6 If you read the documentation, you will notice that (confusingly) command mode is called and commands are called .

Beware.

Moving The Cursor Around

While in command mode, offers a large number of movement commands, some of which it shares with . Here is a subset:

! " %

Key	Moves The Cursor
or Right Arrow	Right one character.
or Left Arrow	Left one character.
or Down Arrow	Down one line.
or Up Arrow	Up one line.
$ (zero)	To the beginning of the current line.
`	To the first non-whitespace character on the current line.
	To the end of the current line.
8	To the beginning of the next word or punctuation character.
.	To the beginning of the next word, ignoring punctuation characters.
	To the beginning of the previous word or punctuation character.
G	To the beginning of the previous word, ignoring punctuation characters.
or : 9 8	Down one page.
or : 26	Up one page.
?	To line ! . For example, "? moves to the first line of the file.
?	To the last line of the file.

Why are the , , , and keys used for cursor movement? Because when was origi-

nally written, not all video terminals had arrow keys, and skilled typists could use regular keyboard keys to move the cursor without ever having to lift their fingers from the keyboard.

Many commands in can be prefixed with a number, as with the “G” command listed above. By prefixing a command with a number, we may specify the number of times a command is to be carried out. For example, the command “5j” causes to move the cursor down five lines.

Basic Editing

Most editing consists of a few basic operations such as inserting text, deleting text, and moving text around by cutting and pasting. , of course, supports all of these operations in its own unique way. also provides a limited form of undo. If we press the “u” key while in command mode, will undo the last change that you made. This will come in handy as we try out some of the basic editing commands.

Appending Text

has several different ways of entering insert mode. We have already used the command to insert text.

Let's go back to our < file for a moment:

```
) A  8  6  B, <
```

If we wanted to add some text to the end of this sentence, we would discover that the command will not do it, since we can't move the cursor beyond the end of the line. provides a command to append text, the sensibly named “a” command. If we move the cursor to the end of the line and type “a”, the cursor will move past the end of the line and will enter insert mode. This will allow us to add some more text:

```
) A  8  6  B, <  P
```

Remember to press the ' key to exit insert mode.

Since we will almost always want to append text to the end of a line, offers a shortcut to move to the end of the current line and start appending. It's the “A” command. Let's try it and add some more lines to our file.

First, we'll move the cursor to the beginning of the line using the “0” (zero) command.

Now we type “A” and add the following lines of text:

```
) A     8      6          B,   < @  8     <
T
T
T   A
T   "
```

Again, press the ' key to exit insert mode.

As we can see, the “A” command is more useful as it moves the cursor to the end of the line before starting insert mode.

Opening A Line

Another way we can insert text is by “opening” a line. This inserts a blank line between two existing lines and enters insert mode. This has two variants:

! " ")

Command	Opens
	The line below the current line.
S	The line above the current line.

We can demonstrate this as follows: place the cursor on “Line 3” then press the key.

```
) A     8      6          B,   < @  8     <
=   *
=   /

=   &
=   #
```

A new line was opened below the third line and we entered insert mode. Exit insert mode by pressing the ' key. Press the key to undo our change.

Press the S key to open the line above the cursor:

```
) A     8      6          B,   < @  8     <
=   *
```

```
=  /
=  &
=  #
```

Exit insert mode by pressing the ' key and undo our change by pressing .

Deleting Text

As we might expect, offers a variety of ways to delete text, all of which contain one of two keystrokes. First, the key will delete a character at the cursor location. may be preceded by a number specifying how many characters are to be deleted. The key is more general purpose. Like , it may be preceded by a number specifying the number of times the deletion is to be performed. In addition, is always followed by a movement command that controls the size of the deletion. Here are some examples:

! " (.

Command	Deletes
	The current character.
/	The current character and the next two characters.
	The current line.
#	The current line and the next four lines.
.	From the current cursor position to the beginning of the next word.
	From the current cursor location to the end of the current line.
$	From the current cursor location to the beginning of the line.
`	From the current cursor location to the first non-whitespace character in the line.
?	From the current line to the end of the file.
*$?	From the current line to the twentieth line of the file.

Place the cursor on the word “It” on the first line of our text. Press the key repeatedly until the rest of the sentence is deleted. Next, press the key repeatedly until the deletion

is undone.

2 6 Real only supports a single level of undo. supports multiple levels.

Let's try the deletion again, this time using the command. Again, move the cursor to the word “It” and press . to delete the word:

```
) A    8    6       B,  < 8    <
=  *
=  /
=  &
=  #
```

Press to delete from the cursor position to the end of the line:

```
) A    8    6       B,  <
=  *
=  /
=  &
=  #
```

Press ? to delete from the current line to the end of the file:

```

```

Press three times to undo the deletion.

Cutting, Copying, And Pasting Text

The command not only deletes text, it also “cuts” text. Each time we use the command the deletion is copied into a paste buffer (think clipboard) that we can later recall with the 6 command to paste the contents of the buffer after the cursor or the : command to paste the contents before the cursor.

The , command is used to “yank” (copy) text in much the same way the command is

used to cut text. Here are some examples combining the , command with various movement commands:

! " - C

Command	Copies
,,	The current line.
#,,	The current line and the next four lines.
,.	From the current cursor position to the beginning of the next word.
,	From the current cursor location to the end of the current line.
,$	From the current cursor location to the beginning of the line.
,`	From the current cursor location to the first non-whitespace character in the line.
,?	From the current line to the end of the file.
,*$?	From the current line to the twentieth line of the file.

Let's try some copy and paste. Place the cursor on the first line of the text and type ,, to copy the current line. Next, move the cursor to the last line (?) and type 6 to paste the line below the current line:

```
) A    8    6       B,  < @ 8    <
=  *
=  /
=  &
=  #
+  ,                 -!   P
```

Just as before, the command will undo our change. With the cursor still positioned on the last line of the file, type : to paste the text above the current line:

```
) A    8    6       B,  < @ 8    <
=  *
=  /
=  &
```

```
+ ,                    -!    P
=  #
```

Try out some of the other , commands in the table above and get to know the behavior of both the 6 and : commands. When you are done, return the file to its original state.

Joining Lines

is rather strict about its idea of a line. Normally, it is not possible to move the cursor to the end of a line and delete the end-of-line character to join one line with the one below it. Because of this, provides a specific command, (not to be confused with , which is for cursor movement) to join lines together.

If we place the cursor on line 3 and type the command, here's what happens:

```
) A    8     6       B,  < @ 8    <
=  *
T   T  A
=  #
```

Search-And-Replace

has the ability to move the cursor to locations based on searches. It can do this on either a single line or over an entire file. It can also perform text replacements with or without confirmation from the user.

Searching Within A Line

The command searches a line and moves the cursor to the next instance of a specified character. For example, the command would move the cursor to the next occurrence of the character “a” within the current line. After performing a character search within a line, the search may be repeated by typing a semicolon.

Searching The Entire File

To move the cursor to the next occurrence of a word or phrase, the 5 command is used. This works the same way as we learned earlier in the program. When you type the 5 command a “/” will appear at the bottom of the screen. Next, type the word or phrase to be searched for, followed by the ' key. The cursor will move to the next location containing the search string. A search may be repeated using the previous search string

with the command. Here's an example:

```
) A     8      6         B,   < @  8     <
=   *
=   /
=   &
=   #
```

Place the cursor on the first line of the file. Type:

```
  T
```

followed by the ' key. The cursor will move to line 2. Next, type and the cursor will move to line 3. Repeating the command will move the cursor down the file until it runs out of matches. While we have so far only used words and phrases for our search patterns, allows the use of , a powerful method of expressing complex text patterns. We will cover regular expressions in some detail in a later chapter.

Global Search-And-Replace

uses an ex command to perform search-and-replace operations (called “substitution” in) over a range of lines or the entire file. To change the word “Line” to “line” for the entire file, we would enter the following command:

```
28  T
```

Let's break this command down into separate items and see what each one does:

! " 2 * !

Item	Meaning
	The colon character starts an ex command.
4	Specifies the range of lines for the operation. 4 is a shortcut meaning from the first line to the last line. Alternately, the range could have been specified "># (since our file is five lines long), or "> which means “from line 1 to the last line in the file.” If the range of lines is omitted, the operation is only performed on the current line.

	Specifies the operation. In this case, substitution (search-and-replace).
5= 5 5	The search pattern and the replacement text.
	This means “global” in the sense that the search-and-replace is performed on every instance of the search string in the line. If omitted, only the first instance of the search string on each line is replaced.

After executing our search-and-replace command our file looks like this:

```
) A    8    6        B,  < @  8    <
  *
  /
  &
  #
```

We can also specify a substitution command with user confirmation. This is done by adding a “c” to the end of the command. For example:

```
28    T
```

This command will change our file back to its previous form; however, before each substitution, stops and asks us to confirm the substitution with this message:

```
 6   8   =   M,5 5 5A5 5`'5`UOE
```

Each of the characters within the parentheses is a possible choice as follows:

! " 3 :

Key	Action
,	Perform the substitution.
	Skip this instance of the pattern.
	Perform the substitution on this and all subsequent instances of the pattern.

A or '	Quit substituting.
	Perform this substitution and then quit. Short for “last.”
> ,	Scroll down and scroll up, respectively. Useful for viewing the context of the proposed substitution.

If you type , , the substitution will be performed, will cause to skip this instance and move on to the next one.

Editing Multiple Files

It's often useful to edit more than one file at a time. You might need to make changes to multiple files or you may need to copy content from one file into another. With we can open multiple files for editing by specifying them on the command line:

```
```

Let's exit our existing vi session and create a new file for editing. Type 8A to exit , saving our modified text. Next, we'll create an additional file in our home directory that we can play with. We'll create the file by capturing some output from the command:

```
                                &
```

Let's edit our old file and our new one with :

```
```

will start up and we will see the first file on the screen:

```
)  A     8     6        B,   < @  8     <
=   *
=   /
=   &
=   #
```

Switching Between Files

To switch from one file to the next, use this ex command:

```
2
```

To move back to the previous file use:

```
2?
```

While we can move from one file to another, enforces a policy that prevents us from switching files if the current file has unsaved changes. To force to switch files and abandon your changes, add an exclamation point (!) to the command.

In addition to the switching method described above, (and some versions of) also provide some ex commands that make multiple files easier to manage. We can view a list of files being edited with the command. Doing so will display a list of the files at the bottom of the display:

```
2
 " 4   _  <  _                  "
 *     _     6 <  _             $
:    'K)'Q    ,6
```

To switch to another buffer (file), type followed by the number of the buffer you wish to edit. For example, to switch from buffer 1 which contains the file < to buffer 2 containing the file 6 < we would type this:

```
2
```

and our screen now displays the second file.

Opening Additional Files For Editing

It's also possible to add files to our current editing session. The ex command (short for “edit”) followed by a filename will open an additional file. Let's end our current editing session and return to the command line.

Start again with just one file:

```

```

To add our second file, enter:

```
2
```

And it should appear on the screen. The first file is still present as we can verify:

```
2
 " Z  _  <  _                  "
 * 4  _    6 <  _              $
:   'K)'Q   ,6
```

---

2 6 You cannot switch to files loaded with the command using either the or K command. To switch files, use the command followed by the buffer number.

Copying Content From One File Into Another

Often while editing multiple files, we will want to copy a portion of one file into another file that we are editing. This is easily done using the usual yank and paste commands we used earlier. We can demonstrate as follows. First, using our two files, switch to buffer 1 (< O by entering:

```
2
```

which should give us this:

```
)  A      8       6          B,   < @  8     <
=   *
=   /
=   &
=   #
```

Next, move the cursor to the first line, and type ,, to yank (copy) the line.

Switch to the second buffer by entering:

```
2
```

The screen will now contain some file listings like this (only a portion is shown here):

```
     /&/0$$
 8       "              /"/"+ *$"# "* $# $! #!
 8       "               !*&$ *$"# "* $% "/ /% &""  66
 8       "             """"*0+ *$"+ $" /" "/ /+  *6
 8       "              *#/+! *$"& "$ $+ *$ "+  #*
 8       "              ""#/* *$"# $# $& "0 &/
 8       "               0*%* *$"# $# $& "0 &/
```

Move the cursor to the first line and paste the line we copied from the preceding file by typing the 6 command:

```
     /&/0$$
+  ,                          -!      P
 8       "              /"/"+ *$"# "* $# $! #!
 8       "               !*&$ *$"# "* $% "/ /% &""  66
 8       "             """"*0+ *$"+ $" /" "/ /+  *6
 8       "              *#/+! *$"& "$ $+ *$ "+  #*
 8       "              ""#/* *$"# $# $& "0 &/
 8       "               0*%* *$"# $# $& "0 &/
```

Inserting An Entire File Into Another

It's also possible to insert an entire file into one that we are editing. To see this in action, let's end our session and start a new one with just a single file:

```

```

We will see our file listing again:

```
    /&/0$$
 8      "            /"/"+ *$"# "* $# $! #!
 8      "             !*&$ *$"# "* $% "/ /% &""  66
 8      "            """"*0+ *$"+ $" /" "/ /+  *6
 8      "            *#/+! *$"& "$ $+ *$ "+  #*
 8      "            ""#/* *$"# $# $& "0 &/
 8      "            0*%* *$"# $# $& "0 &/
```

Move the cursor to the third line, then enter the following ex command:

```
2
```

The command (short for “read”) inserts the specified file before the cursor position. Our screen should now look like this:

```
    /&/0$$
 8      "            /"/"+ *$"# "* $# $! #!
 8      "             !*&$ *$"# "* $% "/ /% &""  66
+ ,                        -!     P
T
T
T   A
T   "
 8      "            """"*0+ *$"+ $" /" "/ /+  *6
 8      "            *#/+! *$"& "$ $+ *$ "+  #*
 8      "            ""#/* *$"# $# $& "0 &/
 8      "            0*%* *$"# $# $& "0 &/
```

Saving Our Work

Like everything else in , there are several different ways to save our edited files. We have already covered the ex command 8 , but there are some others we may also find helpful.

In command mode, typing HH will save the current file and exit . Likewise, the ex command 8A will combine the 8 and A commands into one that will both save the

file and exit.

The 8 command may also specify an optional filename. This acts like “Save As...” For example, if we were editing < and wanted to save an alternate version called "< , we would enter the following:

```
2
```

2 6 While the command above saves the file under a new name, it does not change the name of the file you are editing. As you continue to edit, you will still be editing < , not "< .

Summing Up

With this basic set of skills we can now perform most of the text editing needed to maintain a typical Linux system. Learning to use on a regular basis will pay off in the long run. Since vi-style editors are so deeply embedded in Unix culture, we will see many other programs that have been influenced by its design. is a good example of this influence.

Further Reading

Even with all that we have covered in this chapter, we have barely scratched the surface of what and can do. Here are a couple of on-line resources you can use to continue your journey towards mastery:

1 – A Wikibook from Wikipedia that offers a concise guide to and several of its work-a-likes including . It's available at: http://en.wikibooks.org/wiki/Vi

B 9 - The project has a 570-page book that covers (almost) all of the features in . You can find it at: ftp://ftp.vim.org/pub/vim/doc/book/vimbook-OPL.pdf.

A Wikipedia article on Bill Joy, the creator of .: http://en.wikipedia.org/wiki/Bill_Joy

A Wikipedia article on Bram Moolenaar, the author of : http://en.wikipedia.org/wiki/Bram_Moolenaar

$ 4 2 $"

In this chapter we will look at a seemingly trivial detail — our shell prompt. This examination will reveal some of the inner workings of the shell and the terminal emulator program itself.

Like so many things in Linux, the shell prompt is highly configurable, and while we have pretty much taken it for granted, the prompt is a really useful device once we learn how to control it.

Anatomy Of A Prompt

Our default prompt looks something like this:

```

```

Notice that it contains our username, our hostname and our current working directory, but how did it get that way? Very simply, it turns out. The prompt is defined by an environment variable named :(" (short for “prompt string one”). We can view the contents of :(" with the command:

```
                    3BF
^ ^ ^.^
```

2 6 Don't worry if your results are not exactly the same as the example above. Every Linux distribution defines the prompt string a little differently, some quite exotically.

From the results, we can see that :(" contains a few of the characters we see in our prompt such as the brackets, the at-sign, and the dollar sign, but the rest are a mystery. The astute among us will recognize these as ! like

those we saw in Chapter 7. Here is a partial list of the characters that the treats specially in the prompt string:

! (1 0 + # 7

Sequence	Value Displayed
^	ASCII bell. This makes the computer beep when it is encountered.
^	Current date in day, month, date format. For example, “Mon May 26.”
^	Hostname of the local machine minus the trailing domain name.
^R	Full hostname.
^	Number of jobs running in the current shell session.
^	Name of the current terminal device.
^	A newline character.
^	A carriage return.
^	Name of the shell program.
^	Current time in 24 hour hours:minutes:seconds format.
^)	Current time in 12 hour format.
^	Current time in 12 hour AM/PM format.
^3	Current time in 24 hour hours:minutes format.
^	username of the current user.
^	Version number of the shell.
^;	Version and release numbers of the shell.
^8	Name of the current working directory.
^.	Last part of the current working directory name.
^F	History number of the current command.
^Z	Number of commands entered during this shell session.
^	This displays a “$” character unless we have superuser privileges. In that case, it displays a “#” instead.
^	Signals the start of a series of one or more non-printing characters. This is used to embed non-printing control characters which manipulate the terminal emulator in some way, such as moving the

	cursor or changing text colors.
^	Signals the end of a non-printing character sequence.

Trying Some Alternative Prompt Designs

With this list of special characters, we can change the prompt to see the effect. First, we'll back up the existing prompt string so we can restore it later. To do this, we will copy the existing string into another shell variable that we create ourselves:

```
                    .  $J3BF J
```

We create a new variable called 6 "N and assign the value of :(" to it. We can verify that the string has been copied by using the command:

```
                       3  .
^ ^ ^.^
```

We can restore the original prompt at any time during our terminal session by simply reversing the process:

```
                    BF $J3  .  J
```

Now that we are ready to proceed, let's see what happens if we have an empty prompt string:

```
                    BF $
```

If we assign nothing to the prompt string, we get nothing. No prompt string at all! The prompt is still there, but displays nothing, just as we asked it to. Since this is kind of disconcerting to look at, we'll replace it with a minimal prompt:

```
BF $JK3 J
```

That's better. At least now we can see what we are doing. Notice the trailing space within the double quotes. This provides the space between the dollar sign and the cursor when

the prompt is displayed.

Let's add a bell to our prompt:

```
   BF $JK(K K)K3 J
```

Now we should hear a beep each time the prompt is displayed. This could get annoying, but it might be useful if we needed notification when an especially long-running command has been executed. Note that we included the ^ and ^ sequences. Since the ASCII bell (^) does not “print,” that is, it does not move the cursor, we need to tell so it can correctly determine the length of the prompt.

Next, let's try to make an informative prompt with some hostname and time-of-day information:

```
   BF $JK: K  K3 J
"0 //
```

Adding time-of-day to our prompt will be useful if we need to keep track of when we perform certain tasks. Finally, we'll make a new prompt that is similar to our original:

```
"0 /0                    BF $J/K VK  KW&K3 J
X          C
```

Try out the other sequences listed in the table above and see if you can come up with a brilliant new prompt.

Adding Color

Most terminal emulator programs respond to certain non-printing character sequences to control such things as character attributes (like color, bold text, and the dreaded blinking text) and cursor position. We'll cover cursor position in a little bit, but first we'll look at color.

Terminal Confusion

Back in ancient times, when terminals were hooked to remote computers, there were many competing brands of terminals and they all worked differently. They had different keyboards and they all had different ways of interpreting control information. Unix and Unix-like systems have two rather complex subsystems to deal with the babel of terminal control (called 6 and). If you look in the deepest recesses of your terminal emulator settings you may find a setting for the type of terminal emulation.

In an effort to make terminals speak some sort of common language, the American National Standards Institute (ANSI) developed a standard set of character sequences to control video terminals. Old time DOS users will remember the 3K (@<(U(file that was used to enable interpretation of these codes.

Character color is controlled by sending the terminal emulator an *>#+ embedded in the stream of characters to be displayed. The control code does not "print out" on the display, rather it is interpreted by the terminal as an instruction. As we saw in the table above, the ^ and ^ sequences are used to encapsulate non-printing characters. An ANSI escape code begins with an octal 033 (the code generated by the escape key), followed by an optional character attribute, followed by an instruction. For example, the code to set the text color to normal (attribute = 0), black text is:

^$// $T/$

Here is a table of available text colors. Notice that the colors are divided into two groups, differentiated by the application of the bold character attribute (1) which creates the appearance of "light" colors:

! (" 1 # = 0 #

Sequence	Text Color	Sequence	Text Color
^$// $T/$	Black	^$// "T/$	Dark Gray
^$// $T/"	Red	^$// "T/"	Light Red
^$// $T/*	Green	^$// "T/*	Light Green
^$// $T//	Brown	^$// "T//	Yellow
^$// $T/&	Blue	^$// "T/&	Light Blue
^$// $T/#	Purple	^$// "T/#	Light Purple

^$// $T/+	Cyan	^$// "T/+	Light Cyan
^$// $T/0	Light Grey	^$// "T/0	White

Let's try to make a red prompt. We'll insert the escape code at the beginning:

```
X       C           BF $JK(  K*  (*#   K)/K VK  KW&K3 J
X       C
```

That works, but notice that all the text that we type after the prompt is also red. To fix this, we will add another escape code to the end of the prompt that tells the terminal emulator to return to the previous color:

```
X       C           BF $JK(  K*  (*#   K)/K VK  KW&K3K(K*  (* K) J
X       C
```

That's better!

It's also possible to set the text background color using the codes listed below. The background colors do not support the bold attribute.

! ((1 # = 0 # 9

Sequence	Background Color	Sequence	Background Color
^$// $T&$	Black	^$// $T&&	Blue
^$// $T&"	Red	^$// $T&#	Purple
^$// $T&*	Green	^$// $T&+	Cyan
^$// $T&/	Brown	^$// $T&0	Light Grey

We can create a prompt with a red background by applying a simple change to the first escape code:

```
X       C           BF $JK(  K*  (*#A  K)/K VK  KW&K3K(K*  (* K) J
X       C
```

Try out the color codes and see what you can create!

2 6 Besides the normal (0) and bold (1) character attributes, text may also be given underscore (4), blinking (5), and inverse (7) attributes as well. In the interests of good taste, many terminal emulators refuse to honor the blinking attribute, however.

Moving The Cursor

Escape codes can be used to position the cursor. This is commonly used to provide a clock or some other kind of information at a different location on the screen, such as an upper corner each time the prompt is drawn. Here is a list of the escape codes that position the cursor:

! (- % 1 # =

Escape Code	Action
^$// T R	Move the cursor to line and column
^$// 3	Move the cursor up lines
^$// G	Move the cursor down lines
^$//	Move the cursor forward characters
^$// 9	Move the cursor backward characters
^$// *	Clear the screen and move the cursor to the upper left corner (line 0, column 0)
^$// 1	Clear from the cursor position to the end of the current line
^$//	Store the current cursor position
^$//	Recall the stored cursor position

Using the codes above, we'll construct a prompt that draws a red bar at the top of the screen containing a clock (rendered in yellow text) each time the prompt is displayed. The code for the prompt is this formidable looking string:

```
BF $JK(K* ( K* (*#*RK* (*#A K* (XK* ( # K K* (* K* ( K)
/K VK KW&K3 J
```

Let's take a look at each part of the string to see what it does:

! (2 9) 7 #

Sequence	Action
^	Begins a non-printing character sequence. The purpose of this is to allow to properly calculate the size of the visible prompt. Without an accurate calculation, command line editing features cannot position the cursor correctly.
^$//	Store the cursor position. This is needed to return to the prompt location after the bar and clock have been drawn at the top of the screen. 9 8 '
^$// TR	Move the cursor to the upper left corner, which is line 0, column 0.
^$// $T&"	Set the background color to red.
^$// 1	Clear from the current cursor location (the top left corner) to the end of the line. Since the background color is now red, the line is cleared to that color creating our bar. Note that clearing to the end of the line does not change the cursor position, which remains at the upper left corner.
^$// "T//	Set the text color to yellow.
^	Display the current time. While this is a “printing” element, we still include it in the non-printing portion of the prompt, since we don't want to include the clock when calculating the true size of the displayed prompt.
^$// $	Turn off color. This affects both the text and background.
^$//	Restore the cursor position saved earlier.
^	End the non-printing characters sequence.
X^ ^ ^.C^	Prompt string.

Saving The Prompt

Obviously, we don't want to be typing that monster all the time, so we'll want to store our prompt someplace. We can make the prompt permanent by adding it to our < file. To do so, add these two lines to the file:

```
BF $JK(K*  ( K*  (*#*RK*  (*#A  K*  (XK*  ( #   K K*  (* K*  ( K)
```

```
/K VK  KW&K3 J

    BF
```

Summing Up

Believe it or not, there is much more that can be done with prompts involving shell functions and scripts that we haven't covered here, but this is a good start. Not everyone will care enough to change the prompt, since the default prompt is usually satisfactory. But for those of us who like to tinker, the shell provides the opportunity for many hours of casual fun.

Further Reading

The 9 7 ,)/) from the Linux Documentation Project provides a pretty complete discussion of what the shell prompt can be made to do. It is available at:
http://tldp.org/HOWTO/Bash-Prompt-HOWTO/

Wikipedia has a good article on the ANSI Escape Codes:
http://en.wikipedia.org/wiki/ANSI_escape_code

Part 3 – Common Tasks And Essential Tools

% 2 + & $

If we spend any time in the Linux community, we hear many opinions as to which of the many Linux distributions is “best.” Often, these discussions get really silly, focusing on such things as the prettiness of the desktop background (some people won't use Ubuntu because of its default color scheme!) and other trivial matters.

The most important determinant of distribution quality is the and the vitality of the distribution's support community. As we spend more time with Linux, we see that its software landscape is extremely dynamic. Things are constantly changing. Most of the top-tier Linux distributions release new versions every six months and many individual program updates every day. To keep up with this blizzard of software, we need good tools for .

Package management is a method of installing and maintaining software on the system. Today, most people can satisfy all of their software needs by installing from their Linux distributor. This contrasts with the early days of Linux, when one had to download and compile in order to install software. Not that there is anything wrong with compiling source code; in fact, having access to source code is the great wonder of Linux. It gives us (and everybody else) the ability to examine and improve the system. It's just that having a precompiled package is faster and easier to deal with.

In this chapter, we will look at some of the command line tools used for package management. While all of the major distributions provide powerful and sophisticated graphical programs for maintaining the system, it is important to learn about the command line programs, too. They can perform many tasks that are difficult (or impossible) to do with their graphical counterparts.

Packaging Systems

Different distributions use different packaging systems and as a general rule, a package intended for one distribution is not compatible with another distribution. Most distributions fall into one of two camps of packaging technologies: the Debian “.deb” camp and the Red Hat “.rpm” camp. There are some important exceptions such as Gentoo, Slackware, and Arch, but most others use one of these two basic systems.

! - %A 7 #

Packaging System	Distributions (Partial Listing)
Debian Style (.deb)	Debian, Ubuntu, Linux Mint, Raspbian
Red Hat Style (.rpm)	Fedora, CentOS, Red Hat Enterprise Linux, OpenSUSE

How A Package System Works

The method of software distribution found in the proprietary software industry usually entails buying a piece of installation media such as an “install disk” and then running an “installation wizard” to install a new application on the system.

Linux doesn't work that way. Virtually all software for a Linux system will be found on the Internet. Most of it will be provided by the distribution vendor in the form of and the rest will be available in source code form that can be installed manually. We'll talk about how to install software by compiling source code in chapter 23.

Package Files

The basic unit of software in a packaging system is the . A package file is a compressed collection of files that comprise the software package. A package may consist of numerous programs and data files that support the programs. In addition to the files to be installed, the package file also includes metadata about the package, such as a text description of the package and its contents. Additionally, many packages contain pre- and post-installation scripts that perform configuration tasks before and after the package installation.

Package files are created by a person known as a , often (but not always) an employee of the distribution vendor. The package maintainer gets the software in source code form from the (the author of the program), compiles it, and creates the package metadata and any necessary installation scripts. Often, the package maintainer will apply modifications to the original source code to improve the program's integration with the other parts of the Linux distribution.

Repositories

While some software projects choose to perform their own packaging and distribution, most packages today are created by the distribution vendors and interested third parties. Packages are made available to the users of a distribution in central repositories that may contain many thousands of packages, each specially built and maintained for the distribution.

A distribution may maintain several different repositories for different stages of the software development life cycle. For example, there will usually be a “testing” repository

that contains packages that have just been built and are intended for use by brave souls who are looking for bugs before they are released for general distribution. A distribution will often have a “development” repository where work-in-progress packages destined for inclusion in the distribution's next major release are kept.

A distribution may also have related third-party repositories. These are often needed to supply software that, for legal reasons such as patents or DRM anti-circumvention issues, cannot be included with the distribution. Perhaps the best known case is that of encrypted DVD support, which is not legal in the United States. The third-party repositories operate in countries where software patents and anti-circumvention laws do not apply. These repositories are usually wholly independent of the distribution they support and to use them, one must know about them and manually include them in the configuration files for the package management system.

Dependencies

Programs are seldom “standalone”; rather they rely on the presence of other software components to get their work done. Common activities, such as input/output for example, are handled by routines shared by many programs. These routines are stored in what are called ! , which provide essential services to more than one program. If a package requires a shared resource such as a shared library, it is said to have a . Modern package management systems all provide some method of to ensure that when a package is installed, all of its dependencies are installed, too.

High And Low-level Package Tools

Package management systems usually consist of two types of tools: low-level tools which handle tasks such as installing and removing package files, and high-level tools that perform metadata searching and dependency resolution. In this chapter, we will look at the tools supplied with Debian-style systems (such as Ubuntu and many others) and those used by Red Hat products. While all Red Hat-style distributions rely on the same low-level program (6), they use different high-level tools. For our discussion, we will cover the high-level program , , used by Fedora, Red Hat Enterprise Linux, and CentOS. Other Red Hat-style distributions provide high-level tools with comparable features.

! - " 7 #

Distributions	Low-Level Tools	High-Level Tools
Debian-Style	6	6 > 6
Fedora, Red Hat Enterprise Linux, CentOS	6	,

Common Package Management Tasks

There are many operations that can be performed with the command line package management tools. We will look at the most common. Be aware that the low-level tools also support creation of package files, an activity outside the scope of this book.

In the discussion below, the term “ ” refers to the actual name of a package rather than the term “ ,” which is the name of the file that contains the package.

Finding A Package In A Repository

Using the high-level tools to search repository metadata, a package can be located based on its name or description.

! - (7 #

Style	Command(s)
Debian	6 6 6
Red Hat	,

Example: To search a , repository for the emacs text editor, this command could be used:

```
,
```

Installing A Package From A Repository

High-level tools permit a package to be downloaded from a repository and installed with full dependency resolution.

! - - 7 +

Style	Command(s)
Debian	6 6 6
Red Hat	,

Example: To install the emacs text editor from an 6 repository:

```
6   6  T 6
```

Installing A Package From A Package File

If a package file has been downloaded from a source other than a repository, it can be installed directly (though without dependency resolution) using a low-level tool.

! - 2 7 +

Style	Command(s)
Debian	6
Red Hat	6

Example: If the **<" 0< 0 /!+< 6 package file had been downloaded from a non-repository site, it would be installed this way:

```
6      **<" 0< 0 /!+< 6
```

2 6 Since this technique uses the low-level 6 program to perform the installation, no dependency resolution is performed. If 6 discovers a missing dependency, 6 will exit with an error.

Removing A Package

Packages can be uninstalled using either the high-level or low-level tools. The high-level tools are shown below.

! - 3 7 :

Style	Command(s)
Debian	6
Red Hat	,

Example: To uninstall the emacs package from a Debian-style system:

```
6
```

Updating Packages From A Repository

The most common package management task is keeping the system up-to-date with the latest versions of packages. The high-level tools can perform this vital task in one single step.

! - 4 7 0

Style	Command(s)
Debian	6 6 T 6 6
Red Hat	, 6

Example: To apply any available updates to the installed packages on a Debian-style system:

```
6 6 T 6 6
```

Upgrading A Package From A Package File

If an updated version of a package has been downloaded from a non-repository source, it can be installed, replacing the previous version:

! - 5 7 0

Style	Command(s)
Debian	6
Red Hat	6 2

Example: Updating an existing installation of emacs to the version contained in the package file **<" 0< 0 /!+< 6 on a Red Hat system:

```
6  2     **<" 0<  0  /!+< 6
```

2 6 6 does not have a specific option for upgrading a package versus installing one as 6 does.

Listing Installed Packages

These commands can be used to display a list of all the packages installed on the system:

! - 6 7

Style	Command(s)
Debian	6
Red Hat	6 A

Determining If A Package Is Installed

These low-level tools can be used to display whether a specified package is installed:

! - @ 7 #

Style	Command(s)
Debian	6
Red Hat	6 A

Example: To determine if the emacs package is installed on a Debian style system:

```
6
```

Displaying Info About An Installed Package

If the name of an installed package is known, the following commands can be used to display a description of the package:

! - 7 +

Style	Command(s)
Debian	6 8
Red Hat	,

Example: To see a description of the emacs package on a Debian-style system:

```
6 8
```

Finding Which Package Installed A File

To determine what package is responsible for the installation of a particular file, the following commands can be used:

! - " 7 +

Style	Command(s)
Debian	6
Red Hat	6 A

Example: To see what package installed the 5 5 5 file on a Red Hat system:

```
6 A 5 5 5
```

Summing Up

In the chapters that follow, we will explore many different programs covering a wide range of application areas. While most of these programs are commonly installed by default, we may need to install additional packages if necessary programs are not already installed on our system. With our newfound knowledge (and appreciation) of package management, we should have no problem installing and managing the programs we need.

The Linux Software Installation Myth

People migrating from other platforms sometimes fall victim to the myth that software is somehow difficult to install under Linux and that the variety of packaging schemes used by different distributions is a hindrance. Well, it is a hindrance, but only to proprietary software vendors who wish to distribute binary-only versions of their secret software.

The Linux software ecosystem is based on the idea of open source code. If a program developer releases source code for a program, it is likely that a person associated with a distribution will package the program and include it in their repository. This method ensures that the program is well integrated into the distribution and the user is given the convenience of “one-stop shopping” for software, rather than having to search for each program's web site.

Device drivers are handled in much the same way, except that instead of being separate items in a distribution's repository, they become part of the Linux kernel itself. Generally speaking, there is no such thing as a “driver disk” in Linux. Either the kernel supports a device or it doesn't, and the Linux kernel supports a lot of devices. Many more, in fact, than Windows does. Of course, this is of no consolation if the particular device you need is not supported. When that happens, you need to look at the cause. A lack of driver support is usually caused by one of three things:

+ Since many hardware vendors don't actively support Linux development, it falls upon a member of the Linux community to write the kernel driver code. This takes time.

Not all distributions include every possible device driver. Each distribution builds their own kernels, and since kernels are very configurable (which is what makes it possible to run Linux on everything from wristwatches to mainframes) they may have overlooked a particular device. By locating and downloading the source code for the driver, it is possible for you (yes, you) to compile and install the driver yourself. This process is not overly difficult, but it is rather involved. We'll talk about compiling software in a later chapter.

% + They have neither released source code for a Linux driver, nor have they released the technical documentation for somebody to create one for them. This means that the hardware vendor is trying to keep the programming interfaces to the device a secret. Since we don't want secret devices in our computers, I suggest that you remove the offending hardware and pitch it into the trash with your other useless items.

Further Reading

Spend some time getting to know the package management system for your distribution. Each distribution provides documentation for its package management tools. In addition, here are some more generic sources:

The . ! ?>0< *; chapter on package management provides an overview of package management on Debian systems :
http://www.debian.org/doc/FAQ/ch-pkgtools.en.html

The home page for the RPM project:
http://www.rpm.org

The home page for the YUM project at Duke University:
http://linux.duke.edu/projects/yum/

For a little background, the Wikipedia has an article on metadata:
http://en.wikipedia.org/wiki/Metadata

* &

In previous chapters we've looked at manipulating data at the file level. In this chapter, we will consider data at the device level. Linux has amazing capabilities for handling storage devices, whether physical storage, such as hard disks, or network storage, or virtual storage devices like RAID (Redundant Array of Independent Disks) and LVM (Logical Volume Manager).

However, since this is not a book about system administration, we will not try to cover this entire topic in depth. What we will try to do is introduce some of the concepts and key commands that are used to manage storage devices.

To carry out the exercises in this chapter, we will use a USB flash drive, a CD-RW disc (for systems equipped with a CD-ROM burner) and a floppy disk (again, if the system is so equipped).

We will look at the following commands:

– Mount a file system

– Unmount a file system

– Check and repair a file system

– Partition table manipulator

– Create a file system

– Format a floppy disk

– Write block oriented data directly to a device

M O – Create an ISO 9660 image file

8 M O – Write data to optical storage media

– Calculate an MD5 checksum

Mounting And Unmounting Storage Devices

Recent advances in the Linux desktop have made storage device management extremely

easy for desktop users. For the most part, we attach a device to our system and it “just works.” Back in the old days (say, 2004), this stuff had to be done manually. On non-desktop systems (i.e., servers) this is still a largely manual procedure since servers often have extreme storage needs and complex configuration requirements.

The first step in managing a storage device is attaching the device to the file system tree. This process, called , allows the device to participate with the operating system. As we recall from Chapter 2, Unix-like operating systems, like Linux, maintain a single file system tree with devices attached at various points. This contrasts with other operating systems such as MS-DOS and Windows that maintain separate file system trees for each device (for example ^ , 9 ^ , etc.).

A file named 5 5 (short for “file system table”) lists the devices (typically hard disk partitions) that are to be mounted at boot time. Here is an example 5 5 file from an early Fedora system:

```
=3G'=L5"*      5              &          " "
=3G'=L5        5              &          " *
=3G'=L5        5              &          " *
 6          5  5            6            $ $
  6         5  56             6     L#>   L+*$  $ $
 ,          5 ,             ,             $ $
6           56              6              $ $
=3G'=L(.3:   /   8 6            8 6           $ $
```

Most of the file systems listed in this example file are virtual and are not applicable to our discussion. For our purposes, the interesting ones are the first three:

```
=3G'=L5"*      5              &          " "
=3G'=L5        5              &          " *
=3G'=L5        5              &          " *
```

These are the hard disk partitions. Each line of the file consists of six fields, as follows:

! 2 < < !

Field	Contents	Description
1	Device	Traditionally, this field contains the actual name of a device file associated with the physical device, such as 5 5 " (the first partition of the first detected hard disk). But with today's computers, which have many devices that are hot pluggable (like USB drives),

		many modern Linux distributions associate a device with a text label instead. This label (which is added to the storage media when it is formatted) can be either a simple text label, or a randomly generated UUID (Universally Unique Identifier). This label is read by the operating system when the device is attached to the system. That way, no matter which device file is assigned to the actual physical device, it can still be correctly identified.
2	Mount Point	The directory where the device is attached to the file system tree.
3	File System Type	Linux allows many file system types to be mounted. Most native Linux file systems are Fourth Extended File System (&), but many others are supported, such as FAT16 ( ), FAT32 ( ), NTFS ( ), CD-ROM (%++$), etc.
4	Options	File systems can be mounted with various options. It is possible, for example, to mount file systems as read-only, or to prevent any programs from being executed from them (a useful security feature for removable media).
5	Frequency	A single number that specifies if and when a file system is to be backed up with the 6 command.
6	Order	A single number that specifies in what order file systems should be checked with the command.

Viewing A List Of Mounted File Systems

The command is used to mount file systems. Entering the command without arguments will display a list of the file systems currently mounted:

```
5  5  *   5 ,6    & M 8O
6     56    ,6 6   M 8O
 ,     5 ,  ,6  ,   M 8O
  6    5  56   ,6    6  M 8>  L#>   L+*$O
5  5  #   5    ,6    & M 8O
5  5  "   5    ,6    & M 8O
```

```
 6    5  5    ,6   6  M 8O
     56  5 , 5  5    N    ,6      N    M 8O
  6   5  5  5  5 6 N6 6   ,6  6 N6 6   M 8O
      5 , 5  5   5         ,6       M 8O
5  5  "   5    5    ,6     M 8>    >    >     >
  6 L  >  L#$$>  !>      L  8  O
 8  & 5        5   5       ,6    & M 8>   L"%*<"+!<"<&O
```

The format of the listing is: on $ type $ $ ( ). For example, the first line shows that device 5 5 * is mounted as the root file system, is of type ext4, and is both readable and writable (the option “rw”). This listing also has two interesting entries at the bottom of the list. The next-to-last entry shows a 2 gigabyte SD memory card in a card reader mounted at 5 5 , and the last entry is a network drive mounted at 5 5 .

For our first experiment, we will work with a CD-ROM. First, let's look at a system before a CD-ROM is inserted:

```
5  5  66  5;  ?   6$$ =  ;  $$    5  ,6    & M 8O
6     56    ,6  6    M 8O
,     5 ,   ,6   ,    M 8O
  6    5   56   ,6    6   M 8>   L#>    L+*$O
5  5  "    5     ,6    & M 8O
 6     5   5    ,6    6   M 8O
      56   5 , 5  5     N     ,6       N     M 8O
  6    5   5   5   5 6 N6 6    ,6   6 N6 6    M 8O
```

This listing is from a CentOS 5 system, which is using LVM (Logical Volume Manager) to create its root file system. Like many modern Linux distributions, this system will attempt to automatically mount the CD-ROM after insertion. After we insert the disc, we see the following:

```
5  5  66  5;  ?   6$$ =  ;  $$    5  ,6    & M 8O
6     56    ,6  6    M 8O
,     5 ,   ,6   ,    M 8O
  6    5   56   ,6    6   M 8>   L#>    L+*$O
5  5  "    5     ,6    & M 8O
 6     5   5    ,6    6   M 8O
      56   5 , 5  5     N     ,6       N     M 8O
  6    5   5   5   5 6 N6 6    ,6   6 N6 6    M 8O
5  5     5    5    "<$<"$ !  ,6    %++$ M  >     >     >
```

```
> L#$$O
```

After we insert the disc, we see the same listing as before with one additional entry. At the end of the listing we see that the CD-ROM (which is device 5 5 on this system) has been mounted on 5 5 "<$<"$! , and is type iso9660 (a CD-ROM). For purposes of our experiment, we're interested in the name of the device. When you conduct this experiment yourself, the device name will most likely be different.

' 6 In the examples that follow, it is vitally important that you pay close attention to the actual device names in use on your system and
3

Also note that audio CDs are not the same as CD-ROMs. Audio CDs do not contain file systems and thus cannot be mounted in the usual sense.

Now that we have the device name of the CD-ROM drive, let's unmount the disc and remount it at another location in the file system tree. To do this, we become the superuser (using the command appropriate for our system) and unmount the disc with the
(notice the spelling) command:

```
: 8
        Z
```

The next step is to create a new for the disk. A mount point is simply a directory somewhere on the file system tree. Nothing special about it. It doesn't even have to be an empty directory, though if you mount a device on a non-empty directory, you will not be able to see the directory's previous contents until you unmount the device. For our purposes, we will create a new directory:

```
        Z
```

Finally, we mount the CD-ROM at the new mount point. The option is used to specify the file system type:

```
        Z                DNN*
```

Afterward, we can examine the contents of the CD-ROM via the new mount point:

```
        Z
          Z
```

Notice what happens when we try to unmount the CD-ROM:

```
          Z
   5 5          ,
```

Why is this? The reason is that we cannot unmount a device if the device is being used by someone or some process. In this case, we changed our working directory to the mount point for the CD-ROM, which causes the device to be busy. We can easily remedy the issue by changing the working directory to something other than the mount point:

```
          Z
        Z
```

Now the device unmounts successfully.

Why Unmounting Is Important

If you look at the output of the command, which displays statistics about memory usage, you will see a statistic called “buffers.” Computer systems are designed to go as fast as possible. One of the impediments to system speed is slow devices. Printers are a good example. Even the fastest printer is extremely slow by computer standards. A computer would be very slow indeed if it had to stop and wait for a printer to finish printing a page. In the early days of PCs (before multi-tasking), this was a real problem. If you were working on a spreadsheet or text document, the computer would stop and become unavailable every time you printed. The computer would send the data to the printer as fast as the printer could accept it, but it was very slow since printers don't print very fast. This problem was solved by the advent of the ! , a device containing some RAM memory that would sit between the computer and the printer. With the printer buffer in place, the computer would send the printer output to the buffer and it would quickly be stored in the fast RAM so the computer could go back to

work without waiting. Meanwhile, the printer buffer would slowly the data to the printer from the buffer's memory at the speed at which the printer could accept it.

This idea of buffering is used extensively in computers to make them faster. Don't let the need to occasionally read or write data to or from slow devices impede the speed of the system. Operating systems store data that has been read from, and is to be written to storage devices in memory for as long as possible before actually having to interact with the slower device. On a Linux system for example, you will notice that the system seems to fill up memory the longer it is used. This does not mean Linux is “using“ all the memory, it means that Linux is taking advantage of all the available memory to do as much buffering as it can.

This buffering allows writing to storage devices to be done very quickly, because the writing to the physical device is being deferred to a future time. In the meantime, the data destined for the device is piling up in memory. From time to time, the operating system will write this data to the physical device.

Unmounting a device entails writing all the remaining data to the device so that it can be safely removed. If the device is removed without unmounting it first, the possibility exists that not all the data destined for the device has been transferred. In some cases, this data may include vital directory updates, which will lead to , one of the worst things that can happen on a computer.

Determining Device Names

It's sometimes difficult to determine the name of a device. Back in the old days, it wasn't very hard. A device was always in the same place and it didn't change. Unix-like systems like it that way. Back when Unix was developed, “changing a disk drive” involved using a forklift to remove a washing machine-sized device from the computer room. In recent years, the typical desktop hardware configuration has become quite dynamic and Linux has evolved to become more flexible than its ancestors.

In the examples above we took advantage of the modern Linux desktop's ability to “automagically” mount the device and then determine the name after the fact. But what if we are managing a server or some other environment where this does not occur? How can we figure it out?

First, let's look at how the system names devices. If we list the contents of the 5 directory (where all devices live), we can see that there are lots and lots of devices:

The contents of this listing reveal some patterns of device naming. Here are a few:

! 2 " # . >

Pattern	Device
5 5 D	Floppy disk drives.
5 5 D	IDE (PATA) disks on older systems. Typical motherboards contain two IDE connectors or , each with a cable with two attachment points for drives. The first drive on the cable is called the device and the second is called the device. The device names are ordered such that 5 5 refers to the master device on the first channel, 5 5 is the slave device on the first channel; 5 5 , the master device on the second channel, and so on. A trailing digit indicates the partition number on the device. For example, / 5 " refers to the first partition on the first hard drive on the system while 5 5 refers to the entire drive.
5 5 6D	Printers.
5 5 D	SCSI disks. On modern Linux systems, the kernel treats all disk-like devices (including PATA/SATA hard disks, flash drives, and USB mass storage devices such as portable music players, and digital cameras) as SCSI disks. The rest of the naming system is similar to the older 5 5 D naming scheme described above.
5 5 D	Optical drives (CD/DVD readers and burners).

In addition, we often see symbolic links such as 5 5 , 5 5 , and 5 5 66, , which point to the actual device files, provided as a convenience.

If you are working on a system that does not automatically mount removable devices, you can use the following technique to determine how the removable device is named when it is attached. First, start a real-time view of the 5 5 5 or 5 5 5 , file (you may require superuser privileges for this):

The last few lines of the file will be displayed and then pause. Next, plug in the removable device. In this example, we will use a 16 MB flash drive. Almost immediately, the kernel will notice the device and probe it:

```
   */ "$ $0 #/                / *   8     6   2(G
       N             *
   */ "$ $0 #/                / *           Z"
    "
   */ "$ $0 #/                /   ( (@           2(G -
(
   */ "$ $0 #!                     @K[2@QU
M#O>      /+
   */ "$ $0 #!                 / $ $ $  9     3        ' ,
9            "<$$ :[  $ 3K(@  *
   */ "$ $0 #%                 / $ $ $      /"*+/ #"*  ,
   8          M"+ -GO
   */ "$ $0 #%                 / $ $ $      .    :

   */ "$ $0 #%                 / $ $ $      3
     8
   */ "$ $0 #%                 / $ $ $      /"*+/ #"*  ,
   8          M"+ -GO
   */ "$ $0 #%                 / $ $ $      .    :

   */ "$ $0 #%                 / $ $ $      3
     8
   */ "$ $0 #%                     "
   */ "$ $0 #%                 / $ $ $      3      ( (@

   */ "$ $0 #%                 / $ $ $  3
  /  ,6  $
```

After the display pauses again, press to get the prompt back. The interesting parts of the output are the repeated references to “[sdb]” which matches our expectation of a SCSI disk device name. Knowing this, two lines become particularly illuminating:

```
   */ "$ $0 #%                     "
   */ "$ $0 #%                 / $ $ $      3      ( (@
```

This tells us the device name is 5 5 for the entire device and 5 5 " for the first partition on the device. As we have seen, working with Linux is full of interesting detective work!

(6 Using the 5 5 5 technique is a great way to watch what the system is doing in near real-time.

With our device name in hand, we can now mount the flash drive:

```
   ,          "1         2   3       2  4 -
5  5  *         "#""#&#*   #"!+%&&   %00#"+&  /#4 5
5  5  #         #%+/"%$!  /"000/0+  *&00+&!$  #04 5
5  5  "          "&00+&    "0*00    "**!#!  "/4 5
 6              00+!$!       $    00+!$!   $4 5   5
5  5  "          "##+$       $     "##+$   $4 5   5
```

The device name will remain the same as long as it remains physically attached to the computer and the computer is not rebooted.

Creating New File Systems

Let's say that we want to reformat the flash drive with a Linux native file system, rather than the FAT32 system it has now. This involves two steps: 1. (optional) create a new partition layout if the existing one is not to our liking, and 2. create a new, empty file system on the drive.

' 3 In the following exercise, we are going to format a flash drive. Use a drive that contains nothing you care about because it will be erased! Again,

(" " 7
+ 9 + "
* 7 , + 3

Manipulating Partitions With

The program allows us to interact directly with disk-like devices (such as hard disk drives and flash drives) at a very low level. With this tool we can edit, delete, and create partitions on the device. To work with our flash drive, we must first unmount it (if needed) and then invoke the program as follows:

Notice that we must specify the device in terms of the entire device, not by partition number. After the program starts up, we will see the following prompt:

```
     M        6O
```

Entering an “m” will display the program menu:

```
                  6       ,
            6
           8  6        ,6
     6
           8 6
             8   6 , 9S( 6
  6   6       6
  A   A    8
             8   6 , (
            6       J   ,
             6  ,5    ,
        ,    6
  8   8
                  , M  6       ,O




     M        6O
```

The first thing we want to do is examine the existing partition layout. We do this by entering “p” to print the partition table for the device:

```
     M        6O

9    5   5     "+ -G> "+$$++#+  ,
"      > /"        5     > "$$!  ,
2     L  ,           /" D #"* L "#!0*  ,
```

```
  9    G       (          '      G      @   (,
5  5  "           *      "$$!     "#+$!7     .%#  3)/*
```

In this example, we see a 16 MB device with a single partition (1) that uses 1006 of the available 1008 cylinders on the device. The partition is identified as a Windows 95 FAT32 partition. Some programs will use this identifier to limit the kinds of operation that can be done to the disk, but most of the time it is not critical to change it. However, in the interest of demonstration, we will change it to indicate a Linux partition. To do this, we must first find out what ID is used to identify a Linux partition. In the listing above, we see that the ID “b” is used to specify the existing partition. To see a list of the available partition types, we refer back to the program menu. There we can see the following choice:

```
          8  6        ,6
```

If we enter “l” at the prompt, a large list of possible types is displayed. Among them we see “b” for our existing partition type and “83” for Linux.

Going back to the menu, we see this choice to change a partition ID:

```
           6      J   ,
```

We enter “t” at the prompt enter the new ID:

```
      M        6O
(      6        "
R       M ,6  =           O                        M
       ,     ,6    6        "    !/ M=    O
```

This completes all the changes that we need to make. Up to this point, the device has been untouched (all the changes have been stored in memory, not on the physical device), so we will write the modified partition table to the device and exit. To do this, we enter “w” at the prompt:

```
      M        6O
)  6                      F

         MO          6          <
```

```
.3QK@K?  @  ,                    , 9S( +<
6       > 6                  6
        <
(,        <
```

If we had decided to leave the device unaltered, we could have entered “q” at the prompt, which would have exited the program without writing the changes. We can safely ignore the ominous sounding warning message.

Creating A New File System With

With our partition editing done (lightweight though it might have been) it’s time to create a new file system on our flash drive. To do this, we will use (short for “make file system”), which can create file systems in a variety of formats. To create an ext4 file system on the device, we use the “-t” option to specify the “ext4” system type, followed by the name of the device containing the partition we wish to format:

```
                                    A
  *   "<&$<* M"*     *$$0O
    ,        L
S(  ,6   =
G       B L"$*& M   L$O
         B L"$*& M   L$O
/%$&       > "#+$!
0!$        M#<$$4O                6
             L"
-           ,          L"#%%$0!&
*          6
!"%*       6      6> !"%*          6      6
"%#*       6      6
( 6           6
       !"%/

.
             M"$*&      O
.        6              ,

)        ,    8              ,          , /&
"!$   , > 8                <  2      *                <
```

The program will display a lot of information when ext4 is the chosen file system type. To re-format the device to its original FAT32 file system, specify “vfat” as the file system

type:

```
```

This process of partitioning and formatting can be used anytime additional storage devices are added to the system. While we worked with a tiny flash drive, the same process can be applied to internal hard disks and other removable storage devices like USB hard drives.

Testing And Repairing File Systems

In our earlier discussion of the 5 5 file, we saw some mysterious digits at the end of each line. Each time the system boots, it routinely checks the integrity of the file systems before mounting them. This is done by the program (short for “file system check”). The last number in each entry specifies the order in which the devices are to be checked. In our example above, we see that the root file system is checked first, followed by the and file systems. Devices with a zero as the last digit are not routinely checked.

In addition to checking the integrity of file systems, can also repair corrupt file systems with varying degrees of success, depending on the amount of damage. On Unix-like file systems, recovered portions of files are placed in the 7 directory, located in the root of each file system.

To check our flash drive (which should be unmounted first), we could do the following:

```
   "<&$<! M"/ -   *$"+O
 *    "<&$<! M"/ -   *$"+O
5  5  "      > ""5/%$&     > "++"5"#+$!
```

In my experience, file system corruption is quite rare unless there is a hardware problem, such as a failing disk drive. On most systems, file system corruption detected at boot time will cause the system to stop and direct you to run before continuing.

What The ?

In Unix culture, the word “fsck” is often used in place of a popular word with which it shares three letters. This is especially appropriate, given that you will probably be uttering the aforementioned word if you find yourself in a situation where you are forced to run .

Formatting Floppy Disks

For those of us still using computers old enough to be equipped with floppy diskette drives, we can manage those devices, too. Preparing a blank floppy for use is a two step process. First, we perform a low-level format on the diskette, and then create a file system. To accomplish the formatting, we use the program specifying the name of the floppy device (usually 5 5 $):

```
                                    *
9       > !$     > "!   5    < )     6   , "&&$  G<
        <<<
;   ,   <<<
```

Next, we apply a FAT file system to the diskette with :

```
                                    *
```

Notice that we use the “msdos” file system type to get the older (and smaller) style file allocation tables. After a diskette is prepared, it may be mounted like other devices.

Moving Data Directly To/From Devices

While we usually think of data on our computers as being organized into files, it is also possible to think of the data in “raw” form. If we look at a disk drive, for example, we see that it consists of a large number of “blocks” of data that the operating system sees as directories and files. However, if we could treat a disk drive as simply a large collection of data blocks, we could perform useful tasks, such as cloning devices.

The program performs this task. It copies blocks of data from one place to another. It uses a unique syntax (for historical reasons) and is usually used this way:

```
$                $                ( $                (   $            ))
```

Let's say we had two USB flash drives of the same size and we wanted to exactly copy the first drive to the second. If we attached both drives to the computer and they are assigned to devices 5 5 and 5 5 respectively, we could copy everything on the first drive to the second drive with the following:

```
$      $
```

Alternately, if only the first device were attached to the computer, we could copy its contents to an ordinary file for later restoration or copying:

```
$      $   .
```

' 3 The command is very powerful. Though its name derives from "data definition," it is sometimes called "destroy disk" because users often mistype either the or specifications. # + (((" " (3

Creating CD-ROM Images

Writing a recordable CD-ROM (either a CD-R or CD-RW) consists of two steps; first, constructing an that is the exact file system image of the CD-ROM and second, writing the image file onto the CD-ROM media.

Creating An Image Copy Of A CD-ROM

If we want to make an iso image of an existing CD-ROM, we can use to read all the data blocks off the CD-ROM and copy them to a local file. Say we had an Ubuntu CD and we wanted to make an iso file that we could later use to make more copies. After inserting the CD and determining its device name (we'll assume 5 5), we can make the iso file like so:

```
$      $
```

This technique works for data DVDs as well, but will not work for audio CDs, as they do not use a file system for storage. For audio CDs, look at the command.

Creating An Image From A Collection Of Files

To create an iso image file containing the contents of a directory, we use the program. To do this, we first create a directory containing all the files we wish to include in the image, and then execute the command to create the image file. For example, if we had created a directory called 5 and filled it with files for our CD-ROM, we could create an image file named < with the following command:

```
H Y
```

The “-R” option adds metadata for the : : , which allows the use of long filenames and POSIX style file permissions. Likewise, the “-J” option enables the D , which permit long filenames for Windows.

A Program By Any Other Name...

If you look at on-line tutorials for creating and burning optical media like CD-ROMs and DVDs, you will frequently encounter two programs called and . These programs were part of a popular package called “cdrtools” authored by Jörg Schilling. In the summer of 2006, Mr. Schilling made a license change to a portion of the cdrtools package which, in the opinion of many in the Linux community, created a license incompatibility with the GNU GPL. As a result, a of the cdrtools project was started that now includes replacement programs for and named 8 and , respectively.

Writing CD-ROM Images

After we have an image file, we can burn it onto our optical media. Most of the commands we will discuss below can be applied to both recordable CD-ROM and DVD media.

Mounting An ISO Image Directly

There is a trick that we can use to mount an iso image while it is still on our hard disk and treat it as though it were already on optical media. By adding the “-o loop” option to (along with the required “-t iso9660” file system type), we can mount the image file as though it were a device and attach it to the file system tree:

```
.
DNN*                    .
```

In the example above, we created a mount point named 5 5 N and then mounted the image file < at that mount point. After the image is mounted, it can be treated just as though it were a real CD-ROM or DVD. : !
'

Blanking A Re-Writable CD-ROM

Rewritable CD-RW media needs to be erased or ! before it can be reused. To do this, we can use 8 , specifying the device name for the CD writer and the type of blanking to be performed. The 8 program offers several types. The most minimal (and fastest) is the “fast” type:

```
$          $
```

Writing An Image

To write an image, we again use 8 , specifying the name of the optical media writer device and the name of the image file:

```
$
```

In addition to the device name and image file, 8 supports a very large set of options. Two common ones are “-v” for verbose output, and “-dao”, which writes the disc in mode. This mode should be used if you are preparing a disc for commercial reproduction. The default mode for 8 is , which is useful for recording music tracks.

Summing Up

In this chapter we have looked at the basic storage management tasks. There are, of course, many more. Linux supports a vast array of storage devices and file system schemes. It also offers many features for interoperability with other systems.

Further Reading

Take a look at the man pages of the commands we have covered. Some of them support huge numbers of options and operations. Also, look for on-line tutorials for adding hard drives to your Linux system (there are many) and working with optical media.

Extra Credit

It's often useful to verify the integrity of an iso image that we have downloaded. In most cases, a distributor of an iso image will also supply a . A checksum is the result of an exotic mathematical calculation resulting in a number that represents the content of the target file. If the contents of the file change by even one bit, the resulting checksum will be much different. The most common method of checksum generation uses the # program. When you use # , it produces a unique hexadecimal number:

```
 "
/& /#&0+$ %  0   !# %+ + / %&       <
```

After you download an image, you should run # against it and compare the results with the # value supplied by the publisher.

In addition to checking the integrity of a downloaded file, we can use md5sum to verify newly written optical media. To do this, we first calculate the checksum of the image file and then calculate a checksum for the media. The trick to verifying the media is to limit the calculation to only the portion of the optical media that contains the image. We do this by determining the number of 2048 byte blocks the image contains (optical media is always written in 2048 byte blocks) and reading that many blocks from the media. On some types of media, this is not required. A CD-R written in disc-at-once mode can be checked this way:

```
 "
/& /#&0+$ %  0   !# %+ + / %&    5  5
```

Many types of media, such as DVDs, require a precise calculation of the number of

blocks. In the example below, we check the integrity of the image file < and the disc in the DVD reader 5 5 . Can you figure out how this works?

```
"          #     $        $ *AM     $344 34       J8 J
       6   *AM 66 0  "
```

, 5 +

When it comes to networking, there is probably nothing that cannot be done with Linux. Linux is used to build all sorts of networking systems and appliances, including firewalls, routers, name servers, NAS (Network Attached Storage) boxes and on and on.

Just as the subject of networking is vast, so are the number of commands that can be used to configure and control it. We will focus our attention on just a few of the most frequently used ones. The commands chosen for examination include those used to monitor networks and those used to transfer files. In addition, we are going to explore the program that is used to perform remote logins. This chapter will cover:

6 - Send an ICMP ECHO_REQUEST to network hosts

- Print the route packets trace to a network host

6 - Show / manipulate routing, devices, policy routing and tunnels

- Print network connections, routing tables, interface statistics, masquerade connections, and multicast memberships

6 - Internet file transfer program

8 - Non-interactive network downloader

- OpenSSH SSH client (remote login program)

We're going to assume a little background in networking. In this, the Internet age, everyone using a computer needs a basic understanding of networking concepts. To make full use of this chapter we should be familiar with the following terms:

IP (Internet Protocol) address

Host and domain name

URI (Uniform Resource Identifier)

Please see the "Further Reading" section below for some useful articles regarding these terms.

2 6 Some of the commands we will cover may (depending on your distribution) require the installation of additional packages from your distribution's repositories, and some may require superuser privileges to execute.

Examining And Monitoring A Network

Even if you're not the system administrator, it's often helpful to examine the performance and operation of a network.

6

The most basic network command is 6 . The 6 command sends a special network packet called an ICMP ECHO_REQUEST to a specified host. Most network devices receiving this packet will reply to it, allowing the network connection to be verified.

2 6 It is possible to configure most network devices (including Linux hosts) to ignore these packets. This is usually done for security reasons, to partially obscure a host from a potential attacker. It is also common for firewalls to be configured to block ICMP traffic.

For example, to see if we can reach < (one of our favorite sites ;-), we can use use 6 like this:

```

```

Once started, 6 continues to send packets at a specified interval (default is one second) until it is interrupted:

```
:@K?        <   M++</#<*#$<*"$O #+M!&O  ,       <
+& ,        <       <   M++</#<*#$<*"$O   6N  AL"
  L&/   L"$0
+& ,        <       <   M++</#<*#$<*"$O   6N  AL*
  L&/   L"$!
+& ,        <       <   M++</#<*#$<*"$O   6N  AL/
  L&/   L"$+
+& ,        <       <   M++</#<*#$<*"$O   6N  AL&
  L&/   L"$+
+& ,        <       <   M++</#<*#$<*"$O   6N  AL#
```

```
  L&/    L"$#
+& ,          <         <    M++</#<*#$<*"$O     6N  AL+
  L&/    L"$0

           <   6
+ 6             > +        > $4 6         >      +$"$
     5   5   5     L "$#<+&05"$0<$#*5"$!<""!5$<!*&
```

After it is interrupted (in this case after the sixth packet) by pressing , 6 prints performance statistics. A properly performing network will exhibit zero percent packet loss. A successful “ping” will indicate that the elements of the network (its interface cards, cabling, routing, and gateways) are in generally good working order.

The program (some systems use the similar 6 program instead) displays a listing of all the “hops” network traffic takes to get from the local system to a specified host. For example, to see the route taken to reach < , we would do this:

```

```

The output looks like this:

```
               <    M*"+</&<"!"<&#O> /$   6     > &$  ,
6
 "   6  6<           M"%*<"+!<"<"O  "<$++     "</++     "<0*$
 *  D D D
 /     & "/   $"<        < <   <      <    M+!<!0<"/$<%O  "&<+**
   "&<!!#     "#<"+%
 &  6  /$   $*<        < <   <      <    M+!<!0<"*%<"#&O  "0<+/&
   "0<+*+     "0<!%%
 #  6  +$   $/<        < <   <      <    M+!<!0<"*%<"#!O  "#<%%*
   "#<%!/     "+<*#+
 +  6  /$   $"<  8       ,<  <   <      <    M+!<!0<"/+<#O  **<!/#
   "&<*//     "&<&$#
 0  6  "$   $*<8        < <   <      <    M+!<!0<"*%</&O  "+<"#&
   "/<+$$     "!<!+0
 !     $ / $ "   $"<6       6   <6 <     <      <    M+!<!+<%$<00O
*"<%#"     *"<$0/     *"<##0
 %  6   $ ! $ $   $"<  8,   < ,<     <      <    M+!<!+<!#<"$O
**<%"0     *"<!!&     **<"*+
"$  *$&<0$<"&&<" M*$&<0$<"&&<"O  &/<""$     *"<*&!     *"<*+&
```

```
""    " 6   $ 0 / "<  8,   <      <    M*$&<0$<"%#<%/O  *"<!#0
  * 6   $ $ / "<  8,   <      <    M*$&<0$<*$&<*/!O  "%<##+      "
6   $ 0 / "<  8,   <      <    M*$&<0$<"%#<%/O  "%<+/&
"*    * 6   $ 0 / $<      <      <    M*$&<0$<"%*<"$%O  &"<#!+
&*<!&/     *       $ $ * $<      <      <    M*$&<0$<"%+<*&*O
&/<""#
"/    *                "* "<         /<      <
M*$&<0$<"%#<"**O  &&<*"#     &"<!//     &#<+#!
"&     "   *&"<         /<      <    M*"+<+&<"%&<&*O  &+<!&$
&/</0*     &0<$&"
"#  +&<*0<"+$<"%& M+&<*0<"+$<"%&O  #+<"/0     ##<!!0     #*<!"$
"+          <    M*"+</&<"!"<&#O  &*<0*0     &*<$"+     &"<&/0
```

In the output, we can see that connecting from our test system to < requires traversing sixteen routers. For routers that provided identifying information, we see their hostnames, IP addresses, and performance data, which includes three samples of round-trip time from the local system to the router. For routers that do not provide identifying information (because of router configuration, network congestion, firewalls, etc.), we see asterisks as in the line for hop number 2.

ip

The 6 program is a multi-purpose network configuration tool that makes use of the full range networking features available in modern Linux kernels. It replaces the earlier and now deprecated program. With 6 , we can examine a system's network interfaces and routing table.

```
"     X=SS:G3 1>2:>=S.'QN2:C     +##/+ A       A          2K1KS.K
   6
      5   6     $$ $$ $$ $$ $$ $$     $$ $$ $$ $$ $$ $$
       "*0<$<$<"5!    6
         N          6       N
      +   "5"*!    6
         N          6       N
*     $  XGQS39 3()>-2=)@ 3()>2:>=S.'QN2:C     "#$$ A     6    N
     2:     6        A    "$$$
      5        ** $  #*    !&
       "%*<"+!<"<"&5*&     "%*<"+!<"<*##    6          $
         N          6       N
      +   !$    **      #*   !&5+&    6
         N          6       N
```

In the example above, we see that our test system has two network interfaces. The first,

called , is the ! , a virtual interface that the system uses to “talk to itself” and the second, called $, is the Ethernet interface.

When performing causal network diagnostics, the important things to look for are the presence of the word “UP” in the first line for each interface, indicating that the network interface is enabled, and the presence of a valid IP address in the field on the third line. For systems using DHCP (Dynamic Host Configuration Protocol), a valid IP address in this field will verify that the DHCP is working.

The program is used to examine various network settings and statistics. Through the use of its many options, we can look at a variety of features in our network setup. Using the “-ie” option, we can examine the network interfaces in our system:

```
$   =      6 '       R.    $$ "  $% %  %% +0
          "%*<"+!<"<* G     "%*<"+!<"<*## -    *##<*##<*##<$
      +       !$  *"  %    %  %%+05+& (  6  =
    2: GQS39 3() Q2KK@K? -2=)@ 3()  -)2 "#$$  -     "
    QV 6      */!&!!       $    66   $        $      $
    )V 6      &$/*"0       $    66   $        $      $
            $   A       "$$
    QV  ,    "#/$%!%*" M"&+<$ -GO  )V  ,    *+"$/#*&+ M*&!<% -GO
    -   ,    $$$$    $$$$

    =      6 =    = 6
           "*0<$<$<"  -    *##<$<$<$
      +       "5"*! (  6  R
    2: =SS:G3 1 Q2KK@K?  -)2 "+&/+  -     "
    QV 6      **$!       $    66   $        $      $
    )V 6      **$!       $    66   $        $      $
           $   A       $
    QV  ,    """"&%$ M"$!<! 1GO  )V  ,    """"&%$ M"$!<! 1GO
```

Using the “-r” option will display the kernel’s network routing table. This shows how the network is configured to send packets from network to network:

```
1     @:
9         ?   8 ,    ?                -(( .    8      @

"%*<"+!<"<$  D          *##<*##<*##<$ 2        $ $        $    $
         "%*<"+!<"<" $<$<$<$      2?       $ $        $    $
```

In this simple example, we see a typical routing table for a client machine on a LAN (Local Area Network) behind a firewall/router. The first line of the listing shows the destination "%*<"+!<"<$. IP addresses that end in zero refer to networks rather than individual hosts, so this destination means any host on the LAN. The next field, ? 8 , , is the name or IP address of the gateway (router) used to go from the current host to the destination network. An asterisk in this field indicates that no gateway is needed.

The last line contains the destination . This means any traffic destined for a network that is not otherwise listed in the table. In our example, we see that the gateway is defined as a router with the address of "%*<"+!<"< , which presumably knows what to do with the destination traffic.

Like 6 , the program has many options and we have only looked at a couple. Check out the 6 and man pages for a complete list.

Transporting Files Over A Network

What good is a network unless we can move files across it? There are many programs that move data over networks. We will cover two of them now and several more in later sections.

6

One of the true “classic” programs, 6 gets it name from the protocol it uses, the 7 . FTP is used widely on the Internet for file downloads. Most, if not all, web browsers support it and you often see URIs starting with the protocol 6 55 .

Before there were web browsers, there was the 6 program. 6 is used to communicate with 7 , machines that contain files that can be uploaded and downloaded over a network.

FTP (in its original form) is not secure, because it sends account names and passwords in . This means that they are not encrypted and anyone the network can see them. Because of this, almost all FTP done over the Internet is done by 7 . An anonymous server allows anyone to login using the login name “anonymous” and a meaningless password.

In the example below, we show a typical session with the 6 program downloading an Ubuntu iso image located in the 56 5 N 52 "+<$& directory of the anonymous FTP server :

```
< <
```

```
**$ M   ):  *<$<"O
K    M            O                    !
//" :       6    ,     6   8   <
:   8
*/$ =              <
Q       ,      ,6     2K@V<
2         ,                    <
  6C              .       E       N *A
*#$ 9       ,            ,        <
  6C
*$$ :SQ)                  <              :3(;<
"#$ R                     ,        <
  8  8        " #$$      #$$      0//$0%##* 36  *# $/ #/
"+<$&       6    +&<
**+ 9       ,      S1<
  6C        1
=             ,   8 5    5  59     6
  6C                N *A            NA
            "+<$&       6    +&<                "+<$&       6
   +&<
*$$ :SQ)                  <              :3(;<
"#$ S6      G@K3QU                              "+<$&       6
   +&<    M0//$0%##*  ,   O<
**+           S1<
0//$0%##*  ,                +!<#+      M"$&&"<#  G5 O
  6C     !
```

Here is an explanation of the commands entered during this session:

Command	Meaning
6	Invoke the 6 program and have it connect to the FTP server .
,	Login name. After the login prompt, a password prompt will appear. Some servers will accept a blank password, others will require a password in the form of an email address. In that case, try something like “user@example.com”.
6 5 N 52 "+<$&	Change to the directory on the remote system containing the desired file. Note that on most anonymous FTP servers, the files for public

	downloading are found somewhere under the 6 directory.
	List the directory on the remote system.
9 6	Change the directory on the local system to 59 6 . In the example, the 6 program was invoked when the working directory was . This command changes the working directory to 59 6 .
"+<$& 6 +&<	Tell the remote system to transfer the file "+<$& 6 +&< to the local system. Since the working directory on the local system was changed to 59 6 , the file will be downloaded there.
,	Log off the remote server and end the 6 program session. The commands A and may also be used.

Typing “help” at the “ftp>” prompt will display a list of the supported commands. Using ftp on a server where sufficient permissions have been granted, it is possible to perform many ordinary file management tasks. It’s clumsy, but it does work.

6 – A Better 6

6 is not the only command-line FTP client. In fact, there are many. One of the better (and more popular) ones is 6 by Alexander Lukyanov. It works much like the traditional 6 program, but has many additional convenience features including multiple-protocol support (including HTTP), automatic re-try on failed downloads, background processes, tab completion of path names, and many more.

8

Another popular command-line program for file downloading is 8 . It is useful for downloading content from both web and FTP sites. Single files, multiple files, and even entire sites can be downloaded. To download the first page of < we

could do this:

```
                                        2
  "" $* #"                6 55          <   5     <6 6
         LC I     <6 6J
Q                  <   <<< ++</#<*#$<*"$
                     <   P++</#<*#$<*"$P !$<<<          <
R)):   A        >  8         6    <<< *$$ S1
=         6            5

     XLC                              />"*$          <  15

"" $* #" M"+"<0# -G5 O   I     <6 6J        /"*$
```

The program's many options allow 8 to recursively download, download files in the background (allowing you to log off but continue downloading), and complete the download of a partially downloaded file. These features are well documented in its better-than-average man page.

Secure Communication With Remote Hosts

For many years, Unix-like operating systems have had the ability to be administered remotely via a network. In the early days, before the general adoption of the Internet, there were a couple of popular programs used to log in to remote hosts. These were the and programs. These programs, however, suffer from the same fatal flaw that the 6 program does; they transmit all their communications (including login names and passwords) in cleartext. This makes them wholly inappropriate for use in the Internet age.

To address this problem, a new protocol called SSH (Secure Shell) was developed. SSH solves the two basic problems of secure communication with a remote host. First, it authenticates that the remote host is who it says it is (thus preventing so-called “man in the middle” attacks), and second, it encrypts all of the communications between the local and remote hosts.

SSH consists of two parts. An SSH server runs on the remote host, listening for incoming connections on port 22, while an SSH client is used on the local system to communicate with the remote server.

Most Linux distributions ship an implementation of SSH called OpenSSH from the OpenBSD project. Some distributions include both the client and the server packages by default (for example, Red Hat), while others (such as Ubuntu) only supply the client. To

enable a system to receive remote connections, it must have the O6 ((R package installed, configured and running, and (if the system is either running or is behind a firewall) it must allow incoming network connections on TCP port 22.

(: If you don't have a remote system to connect to but want to try these examples, make sure the S6 ((R package is installed on your system and use as the name of the remote host. That way, your machine will create network connections with itself.

The SSH client program used to connect to remote SSH servers is called, appropriately enough, . To connect to a remote host named , , we would use the client program like so:

```
                                !
)          ,       J       ,  M"%*<"+!<"<&OJ    J
        <
Q(3   ,      6
&"    0    */ "%    /   # "0    +"  / 0   %   <
3   ,       ,   8                    M,  5  OE
```

The first time the connection is attempted, a message is displayed indicating that the authenticity of the remote host cannot be established. This is because the client program has never seen this remote host before. To accept the credentials of the remote host, enter "yes" when prompted. Once the connection is established, the user is prompted for his/her password:

```
.      :       ,      J       , >"%*<"+!<"<&J MQ(3O
    8      <
        , J              6   8
```

After the password is successfully entered, we receive the shell prompt from the remote system:

```
=         (   3   /$ "/ $$ &! *$"+
          ,
```

The remote shell session continues until the user enters the command at the remote shell prompt, thereby closing the remote connection. At this point, the local shell session

resumes and the local shell prompt reappears.

It is also possible to connect to remote systems using a different username. For example, if the local user “me” had an account named “bob” on a remote system, user could log in to the account on the remote system as follows:

```
                                V      !
          , J          6   8
=        (   3   /$ "/ $/ *" *$"+
         ,
```

As stated before, verifies the authenticity of the remote host. If the remote host does not successfully authenticate, the following message appears:

```
                                !

    .3QK@K?  Q'-S)' RS() @9'K)@ @ 3)@SK R3(  R3K?'9F

@) @( :S((@G=' )R3) (S-'SK' @( 9S@K? (S-')R@K? K3()UF
(                  66      ,         8 M
    OF
@        6              Q(3      ,                  <
)       6           Q(3   ,      ,
&"    0     */ "%    /   # "0    +"  / 0   %   <
:            ,    ,              <
3                ,   5    5  5<   5   8 N
     <
S         ,   5    5  5<   5   8 N     "
Q(3       ,          ,              ,        A
     <
R     ,                <
```

This message is caused by one of two possible situations. First, an attacker may be attempting a “man-in-the-middle” attack. This is rare, since everybody knows that alerts the user to this. The more likely culprit is that the remote system has been changed somehow; for example, its operating system or SSH server has been reinstalled. In the interests of security and safety however, the first possibility should not be dismissed out of hand. Always check with the administrator of the remote system when this message occurs.

After it has been determined that the message is due to a benign cause, it is safe to correct the problem on the client side. This is done by using a text editor (perhaps) to remove the obsolete key from the 5 < 5 8 N file. In the example message above, we see this:

```
S       ,  5   5 5<  5   8 N    "
```

This means that line one of the 8 N file contains the offending key. Delete this line from the file, and the program will be able to accept new authentication credentials from the remote system.

Besides opening a shell session on a remote system, also allows us to execute a single command on a remote system. For example, to execute the command on a remote host named , and have the results displayed on the local system:

```
                              !
  8 &J 6  8


-       00##/+   #$0"!&   *+!/#*       $    ""$$+!    "#&#%+

 57      5       *&*#*$   #//$"+
(8 6     "#0*!#+       $    "#0*!#+
```

It’s possible to use this technique in more interesting ways, such as this example in which we perform an ls on the remote system and redirect the output to a file on the local system:

```
                              ! %   % &
  8 &J 6  8
```

Notice the use of the single quotes in the command above. This is done because we do not want the pathname expansion performed on the local machine; rather, we want it to be performed on the remote system. Likewise, if we had wanted the output redirected to a file on the remote machine, we could have placed the redirection operator and the filename within the single quotes:

```
                              ! %    &         %
```

Tunneling With SSH

Part of what happens when you establish a connection with a remote host via SSH is that an is created between the local and remote systems. Normally, this tunnel is used to allow commands typed at the local system to be transmitted safely to the remote system, and for the results to be transmitted safely back. In addition to this basic function, the SSH protocol allows most types of network traffic to be sent through the encrypted tunnel, creating a sort of B7> (Virtual Private Network) between the local and remote systems.

Perhaps the most common use of this feature is to allow X Window system traffic to be transmitted. On a system running an X server (that is, a machine displaying a GUI), it is possible to launch and run an X client program (a graphical application) on a remote system and have its display appear on the local system. It's easy to do; here's an example: Let's say we are sitting at a Linux system called which is running an X server, and we want to run the program on a remote system named , and see the program's graphical output on our local system. We could do this:

```
                                Z      !
        , J            6  8
=       -  ( 6 $! "/ */ "" *$"+
        ,
```

After the command is executed on the remote system, its window appears on the local system. On some systems, you may need to use the "-Y" option rather than the "-X" option to do this.

6 And 6

The OpenSSH package also includes two programs that can make use of an SSH-encrypted tunnel to copy files across the network. The first, 6 (secure copy) is used much like the familiar 6 program to copy files. The most notable difference is that the source or destination pathnames may be preceded with the name of a remote host, followed by a colon character. For example, if we wanted to copy a document named < from our home directory on the remote system, , , to the current working directory on our local system, we could do this:

```
                                ! 2
    , J            6  8
```

```
<                    "$$4 ##!"    #<#1G5    $$ $$
```

As with , you may apply a username to the beginning of the remote host's name if the desired remote host account name does not match that of the local system:

```
                    V      !          2
```

The second SSH file-copying program is 6 which, as its name implies, is a secure replacement for the 6 program. 6 works much like the original 6 program that we used earlier; however, instead of transmitting everything in cleartext, it uses an SSH encrypted tunnel. 6 has an important advantage over conventional 6 in that it does not require an FTP server to be running on the remote host. It only requires the SSH server. This means that any remote machine that can connect with the SSH client can also be used as a FTP-like server. Here is a sample session:

```
                    !
          , <<<
     , J  6   8
  6C
    !<$&     6  /!+<
  6C      1
  6C          M *A        MN
     5   5  5     !<$&     6  /!+<          !<$&
    6  /!+<
5   5  5     !<$&     6  /!+<   "$$4  +%%-G   0<&-G5   $" /#
  6C     !
```

(6 The SFTP protocol is supported by many of the graphical file managers found in Linux distributions. Using either Nautilus (GNOME) or Konqueror (KDE), we can enter a URI beginning with 6 55 into the location bar and operate on files stored on a remote system running an SSH server.

An SSH Client For Windows?

Let's say you are sitting at a Windows machine but you need to log in to your Linux server and get some real work done; what do you do? Get an SSH client program for your Windows box, of course! There are a number of these. The most popular one is probably PuTTY by Simon Tatham and his team. The PuTTY program displays a terminal window and allow a Windows user to open an SSH (or telnet) session on a remote host. The program also provides analogs for the 6 and 6 programs.

PuTTY is available at http://www.chiark.greenend.org.uk/~sgtatham/putty/

Summing Up

In this chapter, we have surveyed the field of networking tools found on most Linux systems. Since Linux is so widely used in servers and networking appliances, there are many more that can be added by installing additional software. But even with the basic set of tools, it is possible to perform many useful network related tasks.

Further Reading

For a broad (albeit dated) look at network administration, the Linux Documentation Project provides the > * E ? :
http://tldp.org/LDP/nag2/index.html

Wikipedia contains many good networking articles. Here are some of the basics:
http://en.wikipedia.org/wiki/Internet_protocol_address
http://en.wikipedia.org/wiki/Host_name
http://en.wikipedia.org/wiki/Uniform_Resource_Identifier

' '

As we have wandered around our Linux system, one thing has become abundantly clear: A typical Linux system has a lot of files! This begs the question, “How do we find things?” We already know that the Linux file system is well organized according to conventions that have been passed down from one generation of Unix-like systems to the next, but the sheer number of files can present a daunting problem.

In this chapter, we will look at two tools that are used to find files on a system. These tools are:

– Find files by name

– Search for files in a directory hierarchy

We will also look at a command that is often used with file-search commands to process the resulting list of files:

– Build and execute command lines from standard input

In addition, we will introduce a couple of commands to assist us in our explorations:

– Change file times

– Display file or file system status

– Find Files The Easy Way

The program performs a rapid database search of pathnames, and then outputs every name that matches a given substring. Say, for example, we want to find all the programs with names that begin with “zip.” Since we are looking for programs, we can assume that the name of the directory containing the programs would end with “bin/”. Therefore, we could try to use this way to find our files:

```
-
```

will search its database of pathnames and output any that contain the string

“bin/zip”:

```
5  5  5B 6
5  5  5B 6
5  5  5B 6  6
5  5  5B 6
5  5  5B 6
5  5  5B 6 6
```

If the search requirement is not so simple, can be combined with other tools such as 6 to design more interesting searches:

```
                    - 0
5  5  B 6*
5  5 B 6*
5  5 B 6*
5  5  B 6
5  5 B 6
5  5  5  B 6
5  5  5 6  B 6
5  5  56   B 6
5  5  56  B 6
5  5  56  B 6
5  5  5  B 6
5  5  5  B 6
5  5  5B 6
5  5  5B 6
5  5  5B 6  6
5  5  5B 6
5  5  5B 6
5  5  5B 6 6
```

The program has been around for a number of years, and there are several different variants in common use. The two most common ones found in modern Linux distributions are and , though they are usually accessed by a symbolic link named . The different versions of have overlapping options sets. Some versions include regular expression matching (which we’ll cover in an upcoming chapter) and wildcard support. Check the man page for to determine which version of is installed.

Where Does The Database Come From?

You may notice that, on some distributions, fails to work just after the system is installed, but if you try again the next day, it works fine. What gives? The database is created by another program named 6 . Usually, it is run periodically as a A ! ; that is, a task performed at regular intervals by the cron daemon. Most systems equipped with run 6 once a day. Since the database is not updated continuously, you will notice that very recent files do not show up when using . To overcome this, it's possible to run the 6 program manually by becoming the superuser and running 6 at the prompt.

– Find Files The Hard Way

While the program can find a file based solely on its name, the program searches a given directory (and its subdirectories) for files based on a variety of attributes. We're going to spend a lot of time with because it has a lot of interesting features that we will see again and again when we start to cover programming concepts in later chapters.

In its simplest use, is given one or more names of directories to search. For example, to produce a list of our home directory:

```

```

On most active user accounts, this will produce a large list. Since the list is sent to standard output, we can pipe the list into other programs. Let's use 8 to count the number of files:

```
                    0
&0$+!
```

Wow, we've been busy! The beauty of is that it can be used to identify files that meet specific criteria. It does this through the (slightly strange) application of , , and . We'll look at the tests first.

Tests

Let’s say that we want a list of directories from our search. To do this, we could add the following test:

```
                    !    0
"+%#
```

Adding the test ,6 limited the search to directories. Conversely, we could have limited the search to regular files with this test:

```
                    !    0
/!0/0
```

Here are the common file type tests supported by find:

! 4

File Type	Description
	Block special device file
	Character special device file
	Directory
	Regular file
	Symbolic link

We can also search by file size and filename by adding some additional tests: Let’s look for all the regular files that match the wildcard pattern “*.JPG” and are larger than one megabyte:

```
                    !      J YBQJ  - 5 O 0
!&$
```

In this example, we add the test followed by the wildcard pattern. Notice how we enclose it in quotes to prevent pathname expansion by the shell. Next, we add the B test followed by the string “+1M”. The leading plus sign indicates that we are looking for files larger than the specified number. A leading minus sign would change the meaning of

the string to be smaller than the specified number. Using no sign means, “match the value exactly.” The trailing letter “M” indicates that the unit of measurement is megabytes. The following characters may be used to specify units:

! 4 " #8 0

Character	Unit
	512-byte blocks. This is the default if no unit is specified.
	Bytes
8	2-byte words
	Kilobytes (units of 1024 bytes)
-	Megabytes (units of 1048576 bytes)
?	Gigabytes (units of 1073741824 bytes)

supports a large number of different tests. Below is a rundown of the common ones. Note that in cases where a numeric argument is required, the same “+” and “-” notation discussed above can be applied:

! 4 (

Test	Description
	Match files or directories whose content or attributes were last modified exactly minutes ago. To specify less than minutes ago, use and to specify more than minutes ago, use .
8	Match files or directories whose contents or attributes were last modified more recently than those of .
	Match files or directories whose contents or attributes were last modified *24 hours ago.
6 ,	Match empty files and directories.
6	Match file or directories belonging to . may be expressed as either a group name or as a numeric group ID.
	Like the test but case insensitive.
	Match files with inode number . This is helpful for finding all the hard links to a particular inode.

	Match files or directories whose contents were last modified minutes ago.
	Match files or directories whose contents were last modified *24 hours ago.
	Match files and directories with the specified wildcard .
8	Match files and directories whose contents were modified more recently than the specified . This is very useful when writing shell scripts that perform file backups. Each time you make a backup, update a file (such as a log), and then use to determine which files have changed since the last update.
	Match file and directories that do not belong to a valid user. This can be used to find files belonging to deleted accounts or to detect activity by attackers.
6	Match files and directories that do not belong to a valid group.
6	Match files or directories that have permissions set to the specified . may be expressed by either octal or symbolic notation.
	Similar to the test. Matches files that share the same inode number as file .
B	Match files of size .
,6	Match files of type .
	Match files or directories belonging to user . The user may be expressed by a username or by a numeric user ID.

This is not a complete list. The man page has all the details.

Operators

Even with all the tests that provides, we may still need a better way to describe the between the tests. For example, what if we needed to determine if all the files and subdirectories in a directory had secure permissions? We would look for all the files with permissions that are not 0600 and the directories with permissions that are not 0700. Fortunately, provides a way to combine tests using

to create more complex logical relationships. To express the aforementioned test, we could do this:

```
                    K4  !          *N** K6    K4  !
    *C** K6
```

Yikes! That sure looks weird. What is all this stuff? Actually, the operators are not that complicated once you get to know them. Here is the list:

! 4 -)

Operator	Description
	Match if the tests on both sides of the operator are true. May be shortened to . Note that when no operator is present, is implied by default.
	Match if a test on either side of the operator is true. May be shortened to .
	Match if the test following the operator is false. May be abbreviated with an exclamation point (F).
M O	Groups tests and operators together to form larger expressions. This is used to control the precedence of the logical evaluations. By default, evaluates from left to right. It is often necessary to override the default evaluation order to obtain the desired result. Even if not needed, it is helpful sometimes to include the grouping characters to improve readability of the command. Note that since the parentheses characters have special meaning to the shell, they must be quoted when using them on the command line to allow them to be passed as arguments to . Usually the backslash character is used to escape them.

With this list of operators in hand, let's deconstruct our command. When viewed from the uppermost level, we see that our tests are arranged as two groupings separated by an operator:

M O M O

This makes sense, since we are searching for files with a certain set of permissions and for directories with a different set. If we are looking for both files and directories, why do

we use instead of ? Because as scans through the files and directories, each one is evaluated to see if it matches the specified tests. We want to know if it is a file with bad permissions a directory with bad permissions. It can't be both at the same time. So if we expand the grouped expressions, we can see it this way:

M 8 6 O M , 8 6 O

Our next challenge is how to test for "bad permissions." How do we do that? Actually we don't. What we will test for is "not good permissions," since we know what "good permissions" are. In the case of files, we define good as 0600 and for directories, as 0700. The expression that will test files for "not good" permissions is:

,6 6 $+$$

and for directories:

,6 6 $0$$

As noted in the table of operators above, the operator can be safely removed, since it is implied by default. So if we put this all back together, we get our final command:

M ,6 6 $+$$ O M ,6
6 $0$$ O

However, since the parentheses have special meaning to the shell, we must escape them to prevent the shell from trying to interpret them. Preceding each one with a backslash character does the trick.

There is another feature of logical operators that is important to understand. Let's say that we have two expressions separated by a logical operator:

6

In all cases, will always be performed; however, the operator will determine if is performed. Here's how it works:

! 4 2 *>.<):

Results of	Operator	is...
True		Always performed
False		Never performed
True		Never performed
False		Always performed

Why does this happen? It's done to improve performance. Take , for example. We know that the expression cannot be true if the result of is

false, so there is no point in performing . Likewise, if we have the expression and the result of is true, there is no point in performing &as we already know that the expression is true.

OK, so it helps it go faster. Why is this important? It's important because we can rely on this behavior to control how actions are performed, as we shall soon see.

Predefined Actions

Let's get some work done! Having a list of results from our command is useful, but what we really want to do is act on the items on the list. Fortunately, allows actions to be performed based on the search results. There are a set of predefined actions and several ways to apply user-defined actions. First let's look at a few of the predefined actions:

! 4 3 7 *

Action	Description
	Delete the currently matching file.
	Perform the equivalent of on the matching file. Output is sent to standard output.
6	Output the full pathname of the matching file to standard output. This is the default action if no other action is specified.
A	Quit once a match has been made.

As with the tests, there are many more actions. See the man page for full details.

In our very first example, we did this:

which produced a list of every file and subdirectory contained within our home directory. It produced a list because the 6 action is implied if no other action is specified. Thus our command could also be expressed as:

We can use to delete files that meet certain criteria. For example, to delete files that

have the file extension ".BAK" (which is often used to designate backup files), we could use this command:

```
    !       % <:X%
```

In this example, every file in the user's home directory (and its subdirectories) is searched for filenames ending in <G31. When they are found, they are deleted.

' 6 It should go without saying that you should when using the action. Always test the command first by substituting the 6 action for to confirm the search results.

Before we go on, let's take another look at how the logical operators affect actions. Consider the following command:

```
    !       % <:X%
```

As we have seen, this command will look for every regular file (,6) whose name ends with <G31 (% <G31J) and will output the relative pathname of each matching file to standard output (6). However, the reason the command performs the way it does is determined by the logical relationships between each of the tests and actions. Remember, there is, by default, an implied relationship between each test and action. We could also express the command this way to make the logical relationships easier to see:

```
    !           % <:X%
```

With our command fully expressed, let's look at how the logical operators affect its execution:

Test/Action	Is Performed Only If...
6	,6 and JD<G31J are true
aD<G31b	,6 is true
,6	Is always performed, since it is the first test/action in an relationship.

Since the logical relationship between the tests and actions determines which of them are performed, we can see that the order of the tests and actions is important. For instance, if we were to reorder the tests and actions so that the 6 action was the first one, the command would behave much differently:

```
          !        % <:X%
```

This version of the command will print each file (the 6 action always evaluates to true) and then test for file type and the specified file extension.

User-Defined Actions

In addition to the predefined actions, we can also invoke arbitrary commands. The traditional way of doing this is with the action. This action works like this:

\] T

where is the name of a command, {} is a symbolic representation of the current pathname, and the semicolon is a required delimiter indicating the end of the command. Here's an example of using to act like the action discussed earlier:

```
    J\]J JTJ
```

Again, since the brace and semicolon characters have special meaning to the shell, they must be quoted or escaped.

It's also possible to execute a user-defined action interactively. By using the action in place of , the user is prompted before execution of each specified command:

```
    !      %   %       %9>% %#%
X   <<< 5   5 5   5   C E              !
 8     "      **& *$$0 "$ *% "! && 5   5 5   5
X   <<< 5   5 5   <   C E              !
 8     "       $ *$"+ $% "% "* #/ 5   5 5   <
```

In this example, we search for files with names starting with the string "foo" and execute the command each time one is found. Using the action prompts the user before the command is executed.

Improving Efficiency

When the action is used, it launches a new instance of the specified command each time a matching file is found. There are times when we might prefer to combine all of the search results and launch a single instance of the command. For example, rather than executing the commands like this:

we may prefer to execute them this way:

thus causing the command to be executed only one time rather than multiple times. There are two ways we can do this. The traditional way, using the external command and the alternate way, using a new feature in itself. We'll talk about the alternate way first.

By changing the trailing semicolon character to a plus sign, we activate the ability of to combine the results of the search into an argument list for a single execution of the desired command. Going back to our example, this:

```
    !       %   %          %9>% %#%
 8     "      **& *$$0 "$ *% "! && 5   5 5  5
 8     "       $ *$"+ $% "% "* #/ 5   5 5  <
```

will execute each time a matching file is found. By changing the command to:

```
    !       %   %          %9>% 5
 8     "      **& *$$0 "$ *% "! && 5   5 5  5
 8     "       $ *$"+ $% "% "* #/ 5   5 5  <
```

we get the same results, but the system only has to execute the command once.

The command performs an interesting function. It accepts input from standard input and converts it into an argument list for a specified command. With our example, we would use it like this:

```
    !       %   %      0
 8     "      **& *$$0 "$ *% "! && 5   5 5  5
 8     "       $ *$"+ $% "% "* #/ 5   5 5  <
```

Here we see the output of the command piped into which, in turn, constructs an argument list for the command and then executes it.

2 6 While the number of arguments that can be placed into a command line is quite large, it's not unlimited. It is possible to create commands that are too long for the shell to accept. When a command line exceeds the maximum length supported by the system, executes the specified command with the maximum number of arguments possible and then repeats this process until standard input is exhausted. To see the maximum size of the command line, execute with the 8 option.

Dealing With Funny Filenames

Unix-like systems allow embedded spaces (and even newlines!) in filenames. This causes problems for programs like that construct argument lists for other programs. An embedded space will be treated as a delimiter, and the resulting command will interpret each space-separated word as a separate argument. To overcome this, and allow the optional use of a as argument separator. A null character is defined in ASCII as the character represented by the number zero (as opposed to, for example, the space character, which is defined in ASCII as the character represented by the number 32). The command provides the action 6 $, which produces null-separated output, and the command has the option, which accepts null separated input. Here's an example:

```
JD< 6 J  6   $ P
```

Using this technique, we can ensure that all files, even those containing embedded spaces in their names, are handled correctly.

A Return To The Playground

It's time to put to some (almost) practical use. We'll create a playground and try out some of what we have learned.

First, let’s create a playground with lots of subdirectories and files:

```
!        9**   **>
!        9**   **>    9:  @>
```

Marvel in the power of the command line! With these two lines, we created a playground directory containing 100 subdirectories each containing 26 empty files. Try that with the GUI!

The method we employed to accomplish this magic involved a familiar command ( ), an exotic shell expansion (braces) and a new command, . By combining with the 6 option (which causes to create the parent directories of the specified paths) with brace expansion, we were able to create 100 subdirectories.

The command is usually used to set or update the access, change, and modify times of files. However, if a filename argument is that of a nonexistent file, an empty file is created.

In our playground, we created 100 instances of a file named 3 . Let’s find them:

```
!      !       %    :%
```

Note that unlike , does not produce results in sorted order. Its order is determined by the layout of the storage device. We can confirm that we actually have 100 instances of the file this way:

```
!      !       %    :% 0
"$$
```

Next, let’s look at finding files based on their modification times. This will be helpful when creating backups or organizing files in chronological order. To do this, we will first create a reference file against which we will compare modification time:

```
!
```

This creates an empty file named 6 and sets its modification time to the current time. We can verify this by using another handy command, , which is a kind of souped-up version of . The command reveals all that the system understands

about a file and its attributes:

```
                                !
      I6  ,     5       6J
  ( B   $         G      $      @S G     &$%+          6 ,
9      !$/ 5*$#"  @      "&*+#$+" =      "
3      M$+&&5  8      O  2    M "$$"5   O   ?    M "$$"5   O
3      *$"+ "$ $! "# "# /%<$$$$$$$$$  $&$$
-    ,  *$"+ "$ $! "# "# /%<$$$$$$$$$  $&$$
       *$"+ "$ $! "# "# /%<$$$$$$$$$  $&$$
```

If we the file again and then examine it with , we will see that the file's times have been updated.:

```
                                !
                                !
      I6  ,     5       6J
  ( B   $         G      $      @S G     &$%+          6 ,
9      !$/ 5*$#"  @      "&*+#$+" =      "
3      M$+&&5  8      O  2    M "$$"5   O   ?    M "$$"5   O
3      *$"+ "$ $! "# */ //<$$$$$$$$$  $&$$
-    ,  *$"+ "$ $! "# */ //<$$$$$$$$$  $&$$
       *$"+ "$ $! "# */ //<$$$$$$$$$  $&$$
```

Next, let's use to update some of our playground files:

```
                                !       !         %     <%
%9>% %#%
```

This updates all files in the playground named G . Next we'll use to identify the updated files by comparing all the files to the reference file 6 :

```
                                !       !            !
```

The results contain all 100 instances of G . Since we performed a on all the files in the playground named G after we updated 6 , they are now "newer" than 6 and thus can be identified with the 8 test.

Finally, let's go back to the bad permissions test we performed earlier and apply it to

6 , :

```
                    !    K4  !          *N** K6    K4
 !          *C** K6
```

This command lists all 100 directories and 2600 files in 6 , (as well as 6 and 6 , itself, for a total of 2702) because none of them meets our definition of “good permissions.” With our knowledge of operators and actions, we can add actions to this command to apply new permissions to the files and directories in our playground:

```
                    !    K4  !          *N**
    *N** %9>% %#% K6    K4  !          *C**
*C** %9>% %#% K6
```

On a day-to-day basis, we might find it easier to issue two commands, one for the directories and one for the files, rather than this one large compound command, but it’s nice to know that we can do it this way. The important point here is to understand how the operators and actions can be used together to perform useful tasks.

Options

Finally, we have the options. The options are used to control the scope of a search. They may be included with other tests and actions when constructing expressions. Here is a list of the most commonly used ones:

! 4 4)

Option	Description
6	Direct to process a directory’s files before the directory itself. This option is automatically applied when the action is specified.
6	Set the maximum number of levels that will descend into a directory tree when performing tests and actions.
6	Set the minimum number of levels that will descend into a directory tree before applying tests and actions.

Direct	not to traverse directories that are mounted on other file systems.
Direct	not to optimize its search based on the assumption that it is searching a Unix-like file system. This is needed when scanning DOS/Windows file systems and CD-ROMs.

Summing Up

It's easy to see that is as simple as is complicated. They both have their uses. Take the time to explore the many features of . It can, with regular use, improve your understanding of Linux files system operations.

Further Reading

The , 6 , , and programs are all part the GNU Project's package. The GNU Project provides a website with extensive on-line documentation, which is quite good and should be read if you are using these programs in high security environments:
http://www.gnu.org/software/findutils/

. ((6 +"

One of the primary tasks of a computer system's administrator is keeping the system's data secure. One way this is done is by performing timely backups of the system's files. Even if you're not a system administrator, it is often useful to make copies of things and to move large collections of files from place to place and from device to device.

In this chapter, we will look at several common programs that are used to manage collections of files. There are the file compression programs:

B 6 – Compress or expand files

B 6* – A block sorting file compressor

The archiving programs:

– Tape archiving utility

B 6 – Package and compress files

And the file synchronization program:

, – Remote file and directory synchronization

Compressing Files

Throughout the history of computing, there has been a struggle to get the most data into the smallest available space, whether that space be memory, storage devices, or network bandwidth. Many of the data services that we take for granted today, such as portable music players, high definition television, or broadband Internet, owe their existence to effective techniques.

Data compression is the process of removing from data. Let's consider an imaginary example. Say we had an entirely black picture file with the dimensions of 100 pixels by 100 pixels. In terms of data storage (assuming 24 bits, or 3 bytes per pixel), the image will occupy 30,000 bytes of storage:

100 * 100 * 3 = 30,000

An image that is all one color contains entirely redundant data. If we were clever, we could encode the data in such a way that we simply describe the fact that we have a block

of 10,000 black pixels. So, instead of storing a block of data containing 30,000 zeros (black is usually represented in image files as zero), we could compress the data into the number 10,000, followed by a zero to represent our data. Such a data compression scheme is called and is one of the most rudimentary compression techniques. Today's techniques are much more advanced and complex but the basic goal remains the same—get rid of redundant data.

(the mathematical techniques used to carry out the compression) fall into two general categories, and . Lossless compression preserves all the data contained in the original. This means that when a file is restored from a compressed version, the restored file is exactly the same as the original, uncompressed version. Lossy compression, on the other hand, removes data as the compression is performed, to allow more compression to be applied. When a lossy file is restored, it does not match the original version; rather, it is a close approximation. Examples of lossy compression are JPEG (for images) and MP3 (for music). In our discussion, we will look exclusively at lossless compression, since most data on computers cannot tolerate any data loss.

gzip

The B 6 program is used to compress one or more files. When executed, it replaces the original file with a compressed version of the original. The corresponding B 6 program is used to restore compressed files to their original, uncompressed form. Here is an example:

```
                                &

8      "        "#0/! *$"+ "$ "& $0 "#    <
                        -

8      "        /*/$ *$"+ "$ "& $0 "#    <   < B
                        -

8      "        "#0/! *$"+ "$ "& $0 "#    <
```

In this example, we create a text file named < from a directory listing. Next, we run B 6 , which replaces the original file with a compressed version named < < B . In the directory listing of <D , we see that the original file has been replaced with the compressed version, and that the compressed version is about one-fifth the size of the original. We can also see that the compressed file has the same permissions and timestamp as the original.

Next, we run the B 6 program to uncompress the file. Afterward, we can see that the compressed version of the file has been replaced with the original, again with the permis-

sions and timestamp preserved.

B 6 has many options. Here are a few:

! 5 8)

Option	Description
	Write output to standard output and keep original files. May also be specified with and .
	Decompress. This causes B 6 to act like B 6 . May also be specified with 6 or 6 .
	Force compression even if a compressed version of the original file already exists. May also be specified with .
	Display usage information. May also be specified with 6 .
	List compression statistics for each file compressed. May also be specified with .
	If one or more arguments on the command line are directories, recursively compress files contained within them. May also be specified with .
	Test the integrity of a compressed file. May also be specified with .
	Display verbose messages while compressing. May also be specified with .
	Set amount of compression. ! is an integer in the range of 1 (fastest, least compression) to 9 (slowest, most compression). The values 1 and 9 may also be expressed as and , respectively. The default value is 6.

Going back to our earlier example:

```
                -
                -          -
< < B           S1
                -          -
```

Here, we replaced the file < with a compressed version named < < B . Next, we tested the integrity of the compressed version, using the and options. Fi-

nally, we decompressed the file back to its original form.

B 6 can also be used in interesting ways via standard input and output:

```
0 - &  -
```

This command creates a compressed version of a directory listing.

The B 6 program, which uncompresses gzip files, assumes that filenames end in the extension < B , so it's not necessary to specify it, as long as the specified name is not in conflict with an existing uncompressed file:

```
-
```

If our goal were only to view the contents of a compressed text file, we could do this:

```
-  0
```

Alternately, there is a program supplied with B 6 , called B , that is equivalent to B 6 with the option. It can be used like the command on B 6 compressed files:

```
-  - 0
```

(6 There is a B program, too. It performs the same function as the pipeline above.

bzip2

The B 6* program, by Julian Seward, is similar to gzip, but uses a different compression algorithm that achieves higher levels of compression at the cost of compression speed. In most regards, it works in the same fashion as B 6 . A file compressed with B 6* is denoted with the extension < B* :

```
                                &

8       "          "#0/! *$"+ "$ "0 "/ #"    <
                          -
                                    -
8       "           *0%* *$"+ "$ "0 "/ #"    <   < B*
                            -          -
```

As we can see, B 6* can be used the same way as B 6 . All the options (except for) that we discussed for B 6 are also supported in B 6* . Note, however, that the compression level option ( ) has a somewhat different meaning to B 6* . B 6* comes with B 6* and B for decompressing files.

B 6* also comes with the B 6* program, which will try to recover damaged < B* files.

Don’t Be Compressive Compulsive

I occasionally see people attempting to compress a file, that has already been compressed with an effective compression algorithm, by doing something like this:

-

Don’t do it. You’re probably just wasting time and space! If you apply compression to a file that is already compressed, you will actually end up with a larger file. This is because all compression techniques involve some overhead that is added to the file to describe the compression. If you try to compress a file that already contains no redundant information, the compression will not result in any savings to offset the additional overhead.

Archiving Files

A common file-management task often used in conjunction with compression is . Archiving is the process of gathering up many files and bundling them together into a single large file. Archiving is often done as a part of system backups. It is also used when old data is moved from a system to some type of long-term storage.

In the Unix-like world of software, the program is the classic tool for archiving files.

Its name, short for , reveals its roots as a tool for making backup tapes. While it is still used for that traditional task, it is equally adept on other storage devices as well. We often see filenames that end with the extension < or < B , which indicate a "plain" tar archive and a gzipped archive, respectively. A tar archive can consist of a group of separate files, one or more directory hierarchies, or a mixture of both. The command syntax works like this:

<<<

where is one of the following operating modes (only a partial list is shown here; see the man page for a complete list):

! 5 " %

Mode	Description
	Create an archive from a list of files and/or directories.
	Extract an archive.
	Append specified pathnames to the end of an archive.
	List the contents of an archive.

uses a slightly odd way of expressing options, so we'll need some examples to show how it works. First, let's re-create our playground from the previous chapter:

```
  !      9**   **>
!      9**   **>    9: @>
```

Next, let's create a tar archive of the entire playground:

```
!       !
```

This command creates a tar archive named 6 , < that contains the entire playground directory hierarchy. We can see that the mode and the option, which is used to specify the name of the archive, may be joined together, and do not require a leading dash. Note, however, that the mode must always be specified first, before any other option.

To list the contents of the archive, we can do this:

```
                                !
```

For a more detailed listing, we can add the (verbose) option:

```
                                !
```

Now, let's extract the playground in a new location. We will do this by creating a new directory named , changing the directory and extracting the tar archive:

```
                                !
6 ,
```

If we examine the contents of 5 56 , , we see that the archive was successfully installed, creating a precise reproduction of the original files. There is one caveat, however: Unless you are operating as the superuser, files and directories extracted from archives take on the ownership of the user performing the restoration, rather than the original owner.

Another interesting behavior of is the way it handles pathnames in archives. The default for pathnames is relative, rather than absolute. does this by simply removing any leading slash from the pathname when creating the archive. To demonstrate, we will re-create our archive, this time specifying an absolute pathname:

```
                                !          !
```

Remember, 56 , will expand into 5 5 56 , when we press the enter key, so we will get an absolute pathname for our demonstration. Next, we will extract the archive as before and watch what happens:

```
                                !
    6 ,
```

```
6 ,
```

Here we can see that when we extracted our second archive, it re-created the directory 5 56 , relative to our current working directory, 5 , not relative to the root directory, as would have been the case with an absolute pathname. This may seem like an odd way for it to work, but it's actually more useful this way, as it allows us to extract archives to any location rather than being forced to extract them to their original locations. Repeating the exercise with the inclusion of the verbose option ( ) will give a clearer picture of what's going on.

Let's consider a hypothetical, yet practical, example of in action. Imagine we want to copy the home directory and its contents from one system to another and we have a large USB hard drive that we can use for the transfer. On our modern Linux system, the drive is "automagically" mounted in the 5 directory. Let's also imagine that the disk has a volume name of G 9 when we attach it. To make the tar archive, we can do the following:

```
< 1
```

After the tar file is written, we unmount the drive and attach it to the second computer. Again, it is mounted at 5 5G 9 . To extract the archive, we do this:

```
*
*5 < 1
```

What's important to see here is that we must first change directory to 5, so that the extraction is relative to the root directory, since all pathnames within the archive are relative.

When extracting an archive, it's possible to limit what is extracted from the archive. For example, if we wanted to extract a single file from an archive, it could be done like this:

```
```

By adding the trailing to the command, will only restore the specified file. Multiple pathnames may be specified. Note that the pathname must be the full, exact rela-

tive pathname as stored in the archive. When specifying pathnames, wildcards are not normally supported; however, the GNU version of (which is the version most often found in Linux distributions) supports them with the 8 option. Here is an example using our previous 6 , < file:

```
                                    !                  %
!              :%
```

This command will extract only files matching the specified pathname including the wildcard D .

is often used in conjunction with to produce archives. In this example, we will use to produce a set of files to include in an archive:

```
                                !          %    :%
  !        %9>% %5%
```

Here we use to match all the files in 6 , named 3 and then, using the action, we invoke in the append mode ( ) to add the matching files to the archive 6 , < .

Using with is a good way of creating ! of a directory tree or an entire system. By using to match files newer than a timestamp file, we could create an archive that only contains files newer than the last archive, assuming that the timestamp file is updated right after each archive is created.

can also make use of both standard input and output. Here is a comprehensive example:

```
                                !          %    :% 0
  $ 0 - & !      -
```

In this example, we used the program to produce a list of matching files and piped them into . If the filename “-” is specified, it is taken to mean standard input or output, as needed. (By the way, this convention of using “-” to represent standard input/output is used by a number of other programs, too). The option (which may also be specified as)) causes to read its list of pathnames from a file rather

than the command line. Lastly, the archive produced by is piped into B 6 to create the compressed archive 6 , < B . The < B extension is the conventional extension given to gzip-compressed tar files. The extension < < B is also used sometimes.

While we used the B 6 program externally to produce our compressed archive, modern versions of GNU support both gzip and bzip2 compression directly with the use of the B and options, respectively. Using our previous example as a base, we can simplify it this way:

```
                    !       %   :% 0    -
!     - +
```

If we had wanted to create a bzip2 compressed archive instead, we could have done this:

```
                    !       %   :% 0
!     - +
```

By simply changing the compression option from B to (and changing the output file's extension to < B to indicate a bzip2 compressed file) we enabled bzip2 compression.

Another interesting use of standard input and output with the command involves transferring files between systems over a network. Imagine that we had two machines running a Unix-like system equipped with and . In such a scenario, we could transfer a directory from a remote system (named , for this example) to our local system:

```
                                  ! %     1     % 0
     ,          b 6  8
9
```

Here we were able to copy a directory named 9 from the remote system , to a directory within the directory named on the local system. How did we do this? First, we launched the program on the remote system using . You will recall that allows us to execute a program remotely on a networked computer and "see" the results on the local system—the standard output pro-

duced on the remote system is sent to the local system for viewing. We can take advantage of this by having create an archive (the mode) and send it to standard output, rather than a file (the option with the dash argument), thereby transporting the archive over the encrypted tunnel provided by to the local system. On the local system, we execute and have it expand an archive (the mode) supplied from standard input (again, the option with the dash argument).

zip

The B 6 program is both a compression tool and an archiver. The file format used by the program is familiar to Windows users, as it reads and writes <B 6 files. In Linux, however, B 6 is the predominant compression program with B 6* being a close second.

In its most basic usage, B 6 is invoked like this:

B 6

For example, to make a zip archive of our playground, we would do this:

```
                    -     !    -    !
```

Unless we include the option for recursion, only the 6 , directory (but none of its contents) is stored. Although the addition of the extension <B 6 is automatic, we will include the file extension for clarity.

During the creation of the zip archive, B 6 will normally display a series of messages like this:

```
    6 ,    5   $*$5    H M      $4O
    6 ,    5   $*$5    U M      $4O
    6 ,    5   $*$5    V M      $4O
    6 ,    5   $!05 M      $4O
    6 ,    5   $!05    ( M      $4O
```

These messages show the status of each file added to the archive. B 6 will add files to the archive using one of two storage methods: Either it will “store” a file without compression, as shown here, or it will “deflate” the file which performs compression. The numeric value displayed after the storage method indicates the amount of compression achieved. Since our playground only contains empty files, no compression is performed on its contents.

Extracting the contents of a zip file is straightforward when using the B 6 program:

```
                    -    !    -
```

One thing to note about B 6 (as opposed to) is that if an existing archive is specified, it is updated rather than replaced. This means that the existing archive is preserved, but new files are added and matching files are replaced.

Files may be listed and extracted selectively from a zip archive by specifying them to B 6 :

```
                    -    !    -    !      *MC    @
3     <<56 ,   <B 6
 =      9    )    K

    $  "$ $# "+ $% *#  6 ,   5   $!05   H

    $           "

                    -    !    -    !      *MC    @
3     <<56 ,   <B 6
 6   6 ,   5   $!05   HE ,  >   > 3  > K  >
           !
      6 ,   5   $!05   H
```

Using the option causes B 6 to merely list the contents of the archive without extracting the file. If no file(s) are specified, B 6 will list all files in the archive. The option can be added to increase the verbosity of the listing. Note that when the archive extraction conflicts with an existing file, the user is prompted before the file is replaced.

Like , B 6 can make use of standard input and output, though its implementation is somewhat less useful. It is possible to pipe a list of filenames to B 6 via the option:

```
                    !       J   :J 0 -  V   : -
```

Here we use to generate a list of files matching the test _ 3_ , and then pipe the list into B 6 , which creates the archive 3<B 6 containing the selected files.

B 6 also supports writing its output to standard output, but its use is limited because very few programs can make use of the output. Unfortunately, the B 6 program does not accept standard input. This prevents B 6 and B 6 from being used together to per-

form network file copying like .

B 6 can, however, accept standard input, so it can be used to compress the output of other programs:

```
                    0 -      -
    M      !$4O
```

In this example we pipe the output of into B 6 . Like , B 6 interprets the trailing dash as “use standard input for the input file.”

The B 6 program allows its output to be sent to standard output when the 6 (for pipe) option is specified:

```
                -       -  0
```

We touched on some of the basic things that B 6 / B 6 can do. They both have a lot of options that add to their flexibility, though some are platform specific to other systems. The man pages for both B 6 and B 6 are pretty good and contain useful examples. However, the main use of these programs is for exchanging files with Windows systems, rather than performing compression and archiving on Linux, where and B 6 are greatly preferred.

Synchronizing Files And Directories

A common strategy for maintaining a backup copy of a system involves keeping one or more directories synchronized with another directory (or directories) located on either the local system (usually a removable storage device of some kind) or a remote system. We might, for example, have a local copy of a website under development and synchronize it from time to time with the “live” copy on a remote web server.

In the Unix-like world, the preferred tool for this task is , . This program can synchronize both local and remote directories by using the , which allows , to quickly detect the differences between two directories and perform the minimum amount of copying required to bring them into sync. This makes , very fast and economical to use, compared to other kinds of copy programs.

, is invoked like this:

,

where and are one of the following:

A local file or directory

A remote file or directory in the form of F G

A remote rsync server specified with a URI of <<F G F G<

Note that either the source or the destination must be a local file. Remote-to-remote copying is not supported.

Let's try , out on some local files. First, let's clean out our directory:

```

```

Next, we'll synchronize the 6 , directory with a corresponding copy in :

```
                    !       !
```

We've included both the option (for archiving—causes recursion and preservation of file attributes) and the option (verbose output) to make a of the 6 , directory within . While the command runs, we will see a list of the files and directories being copied. At the end, we will see a summary message like this:

```
"/#0#% ,          #0!0$ ,    /!0*#!<$$ , 5
  B    /*/$  6   6   $<$*
```

indicating the amount of copying performed. If we run the command again, we will see a different result:

```
                    !       !
         <<<

 **+/# ,          *$ ,    &#/"$<$$ , 5
  B    /*/$  6   6   $<"&
```

Notice that there was no listing of files. This is because , detected that there were no differences between 56 , and 5 56 , , and therefore it didn't need to copy anything. If we modify a file in 6 , and run , again:

```
                          !       *DD    @
                       !       !
            <<<
6 ,    5   $%%5    H
   **+!# ,          &* ,    &#&#&<$$ ,  5
     B    /*/$  6   6   $<"&
```

we see that , detected the change and copied only the updated file.

As a practical example, let's consider the imaginary external hard drive that we used earlier with . If we attach the drive to our system and, once again, it is mounted at 5 5G 9 , we can perform a useful system backup by first creating a directory named 5 6 on the external drive, and then using , to copy the most important stuff from our system to the external drive:

```
                          < 1
                       !
    < 1
```

In this example, we copied the 5 , 5 , and 5 5 directories from our system to our imaginary storage device. We included the option to remove files that may have existed on the backup device that no longer existed on the source device (this is irrelevant the first time we make a backup, but will be useful on subsequent copies). Repeating the procedure of attaching the external drive and running this , command would be a useful (though not ideal) way of keeping a small system backed up. Of course, an alias would be helpful here, too. We could create an alias and add it to our < file to provide this feature:

```
      6LJ    ,          5   5    5  5
5   5G  9  5   6J
```

Now all we have to do is attach our external drive and run the 6 command to do the job.

Using , Over A Network

One of the real beauties of , is that it can be used to copy files over a network. After all, the "r" in , stands for "remote." Remote copying can be done in one of two ways. The first way is with another system that has , installed, along with a remote shell program such as . Let's say we had another system on our local network with a

lot of available hard drive space and we wanted to perform our backup operation using the remote system instead of an external drive. Assuming that it already had a directory named 5 6 where we could deliver our files, we could do this:

```
                    !          $
        ! 2
```

We made two changes to our command to facilitate the network copy. First, we added the L option, which instructs , to use the program as its remote shell. In this way, we were able to use an ssh encrypted tunnel to securely transfer the data from the local system to the remote host. Second, we specified the remote host by prefixing its name (in this case the remote host is named ,) to the destination pathname.

The second way that , can be used to synchronize files over a network is by using an . , can be configured to run as a daemon and listen to incoming requests for synchronization. This is often done to allow mirroring of a remote system. For example, Red Hat Software maintains a large repository of software packages under development for its Fedora distribution. It is useful for software testers to mirror this collection during the testing phase of the distribution release cycle. Since files in the repository change frequently (often more than once a day), it is desirable to maintain a local mirror by periodic synchronization, rather than by bulk copying of the repository. One of these repositories is kept at Georgia Tech; we could mirror it using our local copy of , and their rsync server like this:

```
                !          ! 2  !
            MN
```

In this example, we use the URI of the remote rsync server, which consists of a protocol (, 55), followed by the remote host-name (, < < <), followed by the pathname of the repository.

Summing Up

We've looked at the common compression and archiving programs used on Linux and other Unix-like operating systems. For archiving files, the / B 6 combination is the preferred method on Unix-like systems while B 6 / B 6 is used for interoperability with Windows systems. Finally, we looked at the , program (a personal favorite) which is very handy for efficient synchronization of files and directories across systems.

Further Reading

The man pages for all of the commands discussed here are pretty clear and contain useful examples. In addition, the GNU Project has a good online manual for its version of . It can be found here:
http://www.gnu.org/software/tar/manual/index.html

1 - !"

In the next few chapters, we are going to look at tools used to manipulate text. As we have seen, text data plays an important role on all Unix-like systems, such as Linux. But before we can fully appreciate all of the features offered by these tools, we have to first examine a technology that is frequently associated with the most sophisticated uses of these tools— .

As we have navigated the many features and facilities offered by the command line, we have encountered some truly arcane shell features and commands, such as shell expansion and quoting, keyboard shortcuts, and command history, not to mention the vi editor. Regular expressions continue this "tradition" and may be (arguably) the most arcane feature of them all. This is not to suggest that the time it takes to learn about them is not worth the effort. Quite the contrary. A good understanding will enable us to perform amazing feats, though their full value may not be immediately apparent.

What Are Regular Expressions?

Simply put, regular expressions are symbolic notations used to identify patterns in text. In some ways, they resemble the shell's wildcard method of matching file and pathnames, but on a much grander scale. Regular expressions are supported by many command line tools and by most programming languages to facilitate the solution of text manipulation problems. However, to further confuse things, not all regular expressions are the same; they vary slightly from tool to tool and from programming language to language. For our discussion, we will limit ourselves to regular expressions as described in the POSIX standard (which will cover most of the command line tools), as opposed to many programming languages (most notably 7), which use slightly larger and richer sets of notations.

The main program we will use to work with regular expressions is our old pal, 6 . The name "grep" is actually derived from the phrase "global regular expression print," so we can see that 6 has something to do with regular expressions. In essence, 6 searches text files for the occurrence text matching a specified regular expression and outputs any line containing a match to standard output.

So far, we have used 6 with fixed strings, like so:

```
0 -
```

This will list all the files in the 5 5 directory whose names contain the substring "zip".

The 6 program accepts options and arguments this way:

6

where is a regular expression.

Here is a list of the commonly used 6 options:

! 6)

Option	Description
	Ignore case. Do not distinguish between upper and lower case characters. May also be specified .
	Invert match. Normally, 6 prints lines that contain a match. This option causes 6 to print every line that does not contain a match. May also be specified .
	Print the number of matches (or non-matches if the option is also specified) instead of the lines themselves. May also be specified .
	Print the name of each file that contains a match instead of the lines themselves. May also be specified 8 .
=	Like the option, but print only the names of files that do not contain matches. May also be specified 8 .
	Prefix each matching line with the number of the line within the file. May also be specified -- <
	For multi-file searches, suppress the output of filenames. May also be specified .

In order to more fully explore 6 , let's create some text files to search:

```
                    &
                      &
                    &
                      &

    <          <             <
      <
```

We can perform a simple search of our list of files like this:

```
                    -
    <   B 6*
    <   B 6*
```

In this example, 6 searches all of the listed files for the string B 6 and finds two matches, both in the file < . If we were only interested in the list of files that contained matches rather than the matches themselves, we could specify the option:

```
                     -
    <
```

Conversely, if we wanted only to see a list of the files that did not contain a match, we could do this:

```
                    T -
     <
       <
        <
```

Metacharacters And Literals

While it may not seem apparent, our 6 searches have been using regular expressions all along, albeit very simple ones. The regular expression “bzip” is taken to mean that a match will occur only if the line in the file contains at least four characters and that somewhere in the line the characters “b”, “z”, “i”, and “p” are found in that order, with no other characters in between. The characters in the string “bzip” are all , in that they match themselves. In addition to literals, regular expressions may also in-

clude that are used to specify more complex matches. Regular expression metacharacters consist of the following:

` < \ ] E D 7 M O P ^

All other characters are considered literals, though the backslash character is used in a few cases to create = , as well as allowing the metacharacters to be escaped and treated as literals instead of being interpreted as metacharacters.

2 6 As we can see, many of the regular expression metacharacters are also characters that have meaning to the shell when expansion is performed. When we pass regular expressions containing metacharacters on the command line, it is vital that they be enclosed in quotes to prevent the shell from attempting to expand them.

The Any Character

The first metacharacter we will look at is the dot or period character, which is used to match any character. If we include it in a regular expression, it will match any character in that character position. Here's an example:

```
                    % - %
  B 6*
 B 6*
 B 6*
  B 6
 B 6
  B 6
 6  B 6
6   B 6
6  B 6
6  B 6
 B 6
 B 6
```

We searched for any line in our files that matches the regular expression ".zip". There are a couple of interesting things to note about the results. Notice that the B 6 program was not found. This is because the inclusion of the dot metacharacter in our regular expression increased the length of the required match to four characters, and because the name "zip" only contains three, it does not match. Also, if any files in our lists had contained the file extension <B 6 , they would have been matched as well, because the period character in the file extension would be matched by the "any character," too.

Anchors

The caret (` ) and dollar sign ( ) characters are treated as in regular expressions. This means that they cause the match to occur only if the regular expression is found at the beginning of the line (`) or at the end of the line ( ):

```
                        %[- %
B 6
B 6
B 6  6
B 6
B 6
B 6 6
                        %- 3%
  B 6
 B 6
  B 6
 6  B 6
6   B 6
6  B 6
 B 6
B 6
                        %[- 3%
B 6
```

Here we searched the list of files for the string “zip” located at the beginning of the line, the end of the line, and on a line where it is at both the beginning and the end of the line (i.e., by itself on the line). Note that the regular expression ‘^$’ (a beginning and an end with nothing in between) will match blank lines.

A Crossword Puzzle Helper

Even with our limited knowledge of regular expressions at this point, we can do something useful.

My wife loves crossword puzzles and she will sometimes ask me for help with a particular question. Something like, “What’s a five letter word whose third letter is ‘j’ and last letter is ‘r’ that means...?” This kind of question got me thinking.

Did you know that your Linux system contains a dictionary? It does. Take a look in the 5 5 5 directory and you might find one, or several. The dictionary files located there are just long lists of words, one per line, arranged in alphabetical order. On my system, the 8 file contains just over 98,500

words. To find possible answers to the crossword puzzle question above, we could do this:

```
                                %[    3%
-
```

Using this regular expression, we can find all the words in our dictionary file that are five letters long and have a “j” in the third position and an “r” in the last position.

Bracket Expressions And Character Classes

In addition to matching any character at a given position in our regular expression, we can also match a single character from a specified set of characters by using !
. With bracket expressions, we can specify a set of characters (including characters that would otherwise be interpreted as metacharacters) to be matched. In this example, using a two character set:

```
                          %( )- %
B 6*
B 6*
B 6
```

we match any line that contains the string “bzip” or “gzip”.

A set may contain any number of characters, and metacharacters lose their special meaning when placed within brackets. However, there are two cases in which metacharacters are used within bracket expressions, and have different meanings. The first is the caret (`), which is used to indicate negation; the second is the dash ( ), which is used to indicate a character range.

Negation

If the first character in a bracket expression is a caret (`), the remaining characters are taken to be a set of characters that must not be present at the given character position. We do this by modifying our previous example:

```
                          %([ )- %
 B 6*
```

```
  B 6
  B 6
 6  B 6
6    B 6
6  B 6
6  B 6
 B 6
 B 6
```

With negation activated, we get a list of files that contain the string “zip” preceded by any character except “b” or “g”. Notice that the file B 6 was not found. A negated character set still requires a character at the given position, but the character must not be a member of the negated set.

The caret character only invokes negation if it is the first character within a bracket expression; otherwise, it loses its special meaning and becomes an ordinary character in the set.

Traditional Character Ranges

If we wanted to construct a regular expression that would find every file in our lists beginning with an uppercase letter, we could do this:

```
%[(:<=1G7QRPYXTO?SB\HF+E]WZ@^)%
```

It’s just a matter of putting all 26 uppercase letters in a bracket expression. But the idea of all that typing is deeply troubling, so there is another way:

```
                          %[(: @)%
-31'9';
      :
?')
R'39
:S()
V
V""
V
-31' =S::@'(
K  8   -
K  8   -     9  6
```

By using a three character range, we can abbreviate the 26 letters. Any range of charac-

ters can be expressed this way including multiple ranges, such as this expression that matches all filenames starting with letters and numbers:

```
%[(: @  -* D)%
```

In character ranges, we see that the dash character is treated specially, so how do we actually include a dash character in a bracket expression? By making it the first character in the expression. Consider these two examples:

```
%(: @)%
```

This will match every filename containing an uppercase letter. While:

```
%( :@)%
```

will match every filename containing a dash, or an uppercase “A” or an uppercase “Z”.

POSIX Character Classes

The traditional character ranges are an easily understood and effective way to handle the problem of quickly specifying sets of characters. Unfortunately, they don’t always work. While we have not encountered any problems with our use of 6 so far, we might run into problems using other programs.

Back in Chapter 4, we looked at how wildcards are used to perform pathname expansion. In that discussion, we said that character ranges could be used in a manner almost identical to the way they are used in regular expressions, but here’s the problem:

```
                                (:<=1G7QRPYXTO?SB\HF+E]WZ^@)
5  5   5-31' =S::@'(
5  5   5K  8  -    9  6
5  5   5K  8  -
```

(Depending on the Linux distribution, we will get a different list of files, possibly an empty list. This example is from Ubuntu). This command produces the expected result—a list of only the files whose names begin with an uppercase letter, but:

```
                                (: @)
5  5   5
5  5   5
5  5   5  6  8
5  5   5 6  8
5  5   5
5  5   5    6
5  5   5  6
5  5   5
```

with this command we get an entirely different result (only a partial listing of the results is shown). Why is that? It's a long story, but here's the short version:

Back when Unix was first developed, it only knew about ASCII characters, and this feature reflects that fact. In ASCII, the first 32 characters (numbers 0-31) are control codes (things like tabs, backspaces, and carriage returns). The next 32 (32-63) contain printable characters, including most punctuation characters and the numerals zero through nine. The next 32 (numbers 64-95) contain the uppercase letters and a few more punctuation symbols. The final 31 (numbers 96-127) contain the lowercase letters and yet more punctuation symbols. Based on this arrangement, systems using ASCII used a
that looked like this:

3G 9' ?R@ 1=-KS:[Q()2;.VUH 6A 8 ,B

This differs from proper dictionary order, which is like this:

3 G 9 ' ? R @ 1 = - K S6:A[Q ( ) 2 ;8. V,UBH

As the popularity of Unix spread beyond the United States, there grew a need to support characters not found in U.S. English. The ASCII table was expanded to use a full eight bits, adding characters numbers 128-255, which accommodated many more languages. To support this ability, the POSIX standards introduced a concept called a , which could be adjusted to select the character set needed for a particular location. We can see the language setting of our system using this command:

```
                        3    T:?Q
N2(<2)  !
```

With this setting, POSIX compliant applications will use a dictionary collation order rather than ASCII order. This explains the behavior of the commands above. A character range of 3 H when interpreted in dictionary order includes all of the alphabetic characters except the lowercase "a", hence our results.

To partially work around this problem, the POSIX standard includes a number of character classes which provide useful ranges of characters. They are described in the table be-

low:

Table 19-2: POSIX Character Classes

Character Class	Description
	The alphanumeric characters. In ASCII, equivalent to: 3 H B$ %
8	The same as [:alnum:], with the addition of the underscore (N) character.
6	The alphabetic characters. In ASCII, equivalent to: 3 H B
	Includes the space and tab characters.
	The ASCII control codes. Includes the ASCII characters 0 through 31 and 127.
	The numerals zero through nine.
6	The visible characters. In ASCII, it includes characters 33 through 126.
8	The lowercase letters.
6	The punctuation characters. In ASCII, equivalent to: F_Z 4YJMOD7><5 TXLCE ^^^ NI\P]
6	The printable characters. All the characters in 6 plus the space character.
6	The whitespace characters including space, tab, carriage return, newline, vertical tab, and form feed. In ASCII, equivalent to: ^ ^ ^ ^ ^
66	The uppercase characters.
	Characters used to express hexadecimal numbers. In ASCII, equivalent to: $ %3

Even with the character classes, there is still no convenient way to express partial ranges, such as [A-M].

Using character classes, we can repeat our directory listing and see an improved result:

```
                                ((2    2))
5  5   5-31' =S::@'(
5  5   5K  8  -    9  6
5  5   5K  8  -
```

Remember, however, that this is not an example of a regular expression, rather it is the shell performing pathname expansion. We show it here because POSIX character classes can be used for both.

Reverting To Traditional Collation Order

You can opt to have your system use the traditional (ASCII) collation order by changing the value of the =3K? environment variable. As we saw above, the =3K? variable contains the name of the language and character set used in your locale. This value was originally determined when you selected an installation language as your Linux was installed.

To see the locale settings, use the command:

```
=3K?L  N2(<2)  !
= N )U:'L_  N2(<2)  !_
= NK2-'Q@ L_  N2(<2)  !_
= N)@-'L_  N2(<2)  !_
= N S==3)'L_  N2(<2)  !_
= N-SK')3QUL_  N2(<2)  !_
= N-'((3?'(L_  N2(<2)  !_
= N:3:'QL_  N2(<2)  !_
= NK3-'L_  N2(<2)  !_
= N399Q'((L_  N2(<2)  !_
= N)'=':RSK'L_  N2(<2)  !_
= N-'3(2Q'-'K)L_  N2(<2)  !_
= N@9'K)@ @ 3)@SKL_  N2(<2)  !_
= N3==L
```

To change the locale to use the traditional Unix behaviors, set the =3K? variable to :S(@V:

```
T:?Q$BSFPZ
```

Note that this change converts the system to use U.S. English (more specifically, ASCII) for its character set, so be sure if this is really what you want.

You can make this change permanent by adding this line to you your <
file:

6 =3K?L:S(@V

POSIX Basic Vs. Extended Regular Expressions

Just when we thought this couldn't get any more confusing, we discover that POSIX also splits regular expression implementations into two kinds: ! H9:1I and H1:1I. The features we have covered so far are supported by any application that is POSIX compliant and implements BRE. Our 6 program is one such program.

What's the difference between BRE and ERE? It's a matter of metacharacters. With BRE, the following metacharacters are recognized:

` < D

All other characters are considered literals. With ERE, the following metacharacters (and their associated functions) are added:

M O \ ] E 7 P

However (and this is the fun part), the "(", ")", "{", and "}" characters are treated as metacharacters in BRE they are escaped with a backslash, whereas with ERE, preceding any metacharacter with a backslash causes it to be treated as a literal. Any weirdness that comes along will be covered in the discussions that follow.

Since the features we are going to discuss next are part of ERE, we are going to need to use a different 6 . Traditionally, this has been performed by the 6 program, but the GNU version of 6 also supports extended regular expressions when the ' option is used.

POSIX

During the 1980's, Unix became a very popular commercial operating system, but by 1988, the Unix world was in turmoil. Many computer manufacturers had licensed the Unix source code from its creators, AT&T, and were supplying various versions of the operating system with their systems. However, in their efforts to create product differentiation, each manufacturer added proprietary changes and extensions. This started to limit the compatibility of the software. As always with

proprietary vendors, each was trying to play a winning game of “lock-in” with their customers. This dark time in the history of Unix is known today as “ 9 8 .”

Enter the IEEE (Institute of Electrical and Electronics Engineers). In the mid-1980s, the IEEE began developing a set of standards that would define how Unix (and Unix-like) systems would perform. These standards, formally known as IEEE 1003, define the (APIs), shell and utilities that are to be found on a standard Unix-like system. The name “POSIX,” which stands for 7 !) # + (with the “X” added to the end for extra snappiness), was suggested by Richard Stallman (yes, Richard Stallman), and was adopted by the IEEE.

Alternation

The first of the extended regular expression features we will discuss is called , which is the facility that allows a match to occur from among a set of expressions. Just as a bracket expression allows a single character to match from a set of specified characters, alternation allows matches from a set of strings or other regular expressions.

To demonstrate, we’ll use 6 in conjunction with . First, let’s try a plain old string match:

```
                    J:::J 0    :::
333
                    J<<<J 0    :::
```

A pretty straightforward example, in which we pipe the output of into 6 and see the results. When a match occurs, we see it printed out; when no match occurs, we see no results.

Now we’ll add alternation, signified by the vertical-bar metacharacter:

```
                    J:::J 0    G %:::0<<<%
333
                    J<<<J 0    G %:::0<<<%
GGG
                    J===J 0    G %:::0<<<%
```

Here we see the regular expression J333PGGGJ, which means “match either the string 333 or the string GGG” Notice that since this is an extended feature, we added the ' option to 6 (though we could have just used the 6 program instead), and we enclosed the regular expression in quotes to prevent the shell from interpreting the vertical-bar metacharacter as a pipe operator. Alternation is not limited to two choices:

```
                    J:::J 0      G %:::0<<<0===%
333
```

To combine alternation with other regular expression elements, we can use () to separate the alternation:

```
                    G  %[4 -0 -0-  6%
```

This expression will match the filenames in our lists that start with either “bz”, “gz”, or “zip”. Had we left off the parentheses, the meaning of this regular expression :

```
                    G  %[ -0 -0-  %
```

changes to match any filename that begins with “bz” “gz” “zip”.

Quantifiers

Extended regular expressions support several ways to specify the number of times an element is matched.

? - Match An Element Zero Or One Time

This quantifier means, in effect, “Make the preceding element optional.” Let’s say we wanted to check a phone number for validity and we considered a phone number to be valid if it matched either of these two forms:

M O

where “ ” is a numeral. We could construct a regular expression like this:

`^ME $ % $ % $ % ^OE $ % $ % $ % $ % $ % $ % $ %

In this expression, we follow the parentheses characters with question marks to indicate that they are to be matched zero or one time. Again, since the parentheses are normally

metacharacters (in ERE), we precede them with backslashes to cause them to be treated as literals instead.

Let's try it:

```
                              J4"""6     A"NCJ 0       G %[K4_(* D)(* D)(* D)
K6_ (* D)(* D)(* D) (* D)(* D)(* D)(* D)3%
M###O "*/ &#+0
                              J"""       A"NCJ 0       G %[K4_(* D)(* D)(* D)K6
_ (* D)(* D)(* D) (* D)(* D)(* D)(* D)3%
### "*/ &#+0
                              J:::       A"NCJ 0       G %[K4_(* D)(* D)(* D)K6
_ (* D)(* D)(* D) (* D)(* D)(* D)(* D)3%
```

Here we see that the expression matches both forms of the phone number, but does not match one containing non-numeric characters.

* - Match An Element Zero Or More Times

Like the E metacharacter, the D is used to denote an optional item; however, unlike the E, the item may occur any number of times, not just once. Let's say we wanted to see if a string was a sentence; that is, it starts with an uppercase letter, then contains any number of upper and lowercase letters and spaces, and ends with a period. To match this (very crude) definition of a sentence, we could use a regular expression like this:

```
66       66       8      D^<
```

The expression consists of three items: a bracket expression containing the 66 character class, a bracket expression containing both the 66 and 8 character classes and a space, and a period escaped with a backslash. The second element is trailed with an D metacharacter, so that after the leading uppercase letter in our sentence, any number of upper and lowercase letters and spaces may follow it and still match:

```
                              J+         J 0       G %((2     2))((2     2)(
2     2) ) K %
)    8    <
                              J+    W    J 0       G %((2     2))((2     2)(
2     2) ) K %
)    .    <
                              J          J 0       G %((2     2))((2     2
)(2     2) ) K %
```

The expression matches the first two tests, but not the third, since it lacks the required leading uppercase character and trailing period.

+ - Match An Element One Or More Times

The 7 metacharacter works much like the D, except it requires at least one instance of the preceding element to cause a match. Here is a regular expression that will only match lines consisting of groups of one or more alphabetic characters separated by single spaces:

`M 6 7 EO7

```
                    J+      J 0      G %[4((2    2))5 _653%
)
                    J    J 0      G %[4((2    2))5 _653%

                    J   DJ 0      G %[4((2    2))5 _653%
                    J     J 0      G %[4((2    2))5 _653%
```

We see that this expression does not match the line “a b 9”, because it contains a non-alphabetic character; nor does it match “abc d”, because more than one space character separates the characters “c” and “d”.

{ } - Match An Element A Specific Number Of Times

The \ and] metacharacters are used to express minimum and maximum numbers of required matches. They may be specified in four possible ways:

! 6 (# > !) %

Specifier	Meaning
\ ]	Match the preceding element if it occurs exactly times.
\ >]	Match the preceding element if it occurs at least times, but no more than times.
\ >]	Match the preceding element if it occurs or more times.
\>]	Match the preceding element if it occurs no more than times.

Going back to our earlier example with the phone numbers, we can use this method of specifying repetitions to simplify our original regular expression from:

```
`^ME $ %  $ %  $ % ^OE   $ %  $ %  $ %   $ %  $ %  $ %  $ %
```

to:

```
`^ME $ % \/]^OE   $ % \/]  $ % \&]
```

Let’s try it:

```
                          J4"""6     A"NCJ 0       G %[K4_(* D)9 >K6_ (*
D)9 > (* D)9A>3%
M###O "*/ &#+0
                          J"""       A"NCJ 0       G %[K4_(* D)9 >K6_ (* D)
9 > (* D)9A>3%
### "*/ &#+0
                          J"""""     A"NCJ 0       G %[K4_(* D)9 >K6_ (* D
)9 > (* D)9A>3%
```

As we can see, our revised expression can successfully validate numbers both with and without the parentheses, while rejecting those numbers that are not properly formatted.

Putting Regular Expressions To Work

Let’s look at some of the commands we already know and see how they can be used with regular expressions.

Validating A Phone List With 6

In our earlier example, we looked at single phone numbers and checked them for proper formatting. A more realistic scenario would be checking a list of numbers instead, so let’s make a list. We’ll do this by reciting a magical incantation to the command line. It will be magic because we have not covered most of the commands involved, but worry not. We will get there in future chapters. Here is the incantation:

```
                                9    *>#        J439H:?1SO2*2 >6 39H:?1S
O2*2 > 39H:?1SO2*2A>J &&            #
```

This command will produce a file named 6 < containing ten phone numbers. Each time the command is repeated, another ten numbers are added to the list. We can also change the value "$ near the beginning of the command to produce more or fewer phone numbers. If we examine the contents of the file, however, we see we have a problem:

```
M*/*O *%! **+#
M+*&O /!" "$0!
M#&$O "*+ "%!$
M!0&O "+/ *!!#
M*!+O *#& *!+$
M*%*O "$! #"!
M"*%O && "/0%
M&#!O *0/ "+&*
M+!+O *%% !*+!
M"%!O /$0 *&&$
```

Some of the numbers are malformed, which is perfect for our purposes, since we will use 6 to validate them.

One useful method of validation would be to scan the file for invalid numbers and display the resulting list:

```
                                G  %[K4(* D)9 >K6 (* D)9 > (* D)9A>3%
6       <
M*%*O "$! #"!
M"*%O && "/0%
```

Here we use the option to produce an inverse match so that we will only output the lines in the list that do not match the specified expression. The expression itself includes the anchor metacharacters at each end to ensure that the number has no extra characters at either end. This expression also requires that the parentheses be present in a valid number, unlike our earlier phone number example.

Finding Ugly Filenames With

The command supports a test based on a regular expression. There is an important consideration to keep in mind when using regular expressions in versus 6 . Whereas 6 will print a line when the line a string that matches an expression, requires that the pathname the regular expression. In the following example, we will use with a regular expression to find every pathname that contains any character that is not a member of the following set:

N<5$ % B3 H

Such a scan would reveal pathnames that contain embedded spaces and other potentially offensive characters:

```
% ([ . * D -: @) %
```

Due to the requirement for an exact match of the entire pathname, we use <D at both ends of the expression to match zero or more instances of any character. In the middle of the expression, we use a negated bracket expression containing our set of acceptable pathname characters.

Searching For Files With

The program supports both basic (the 6 option) and extended (the option) regular expressions. With it, we can perform many of the same operations that we performed earlier with our files:

```
% 4 -0 -0- 6%
5 5 B
5 5 B 6
5 5 B
5 5 B 6
5 5 B
5 5 B 6
5 5 B 6
5 5 B 6*
5 5 B 6*
5 5 B
5 5 B
5 5 B
5 5 B 6
5 5 5B 6
5 5 5B 6
5 5 5B 6 6
5 5 5B 6
5 5 5B 6
5 5 5B 6 6
```

Using alternation, we perform a search for pathnames that contain either 5 B , 5 B , or 5 5B 6 .

Searching For Text With And

and both share the same method of searching for text. Pressing the 5 key followed by a regular expression will perform a search. If we use to view our 6 < file:

```

```

then search for our validation expression:

```
M*/*O *%! **+#
M+*&O /!" "$0!
M#&$O "*+ "%!$
M!0&O "+/ *!!#
M*!+O *#& *!+$
M*%*O "$! #"!
M"*%O && "/0%
M&#!O *0/ "+&*
M+!+O *%% !*+!
M"%!O /$0 *&&$



 [K4(* D)9 >K6 (* D)9 > (* D)9A>3
```

will highlight the strings that match, leaving the invalid ones easy to spot:

```
M*/*O *%! **+#
M+*&O /!" "$0!
M#&$O "*+ "%!$
M!0&O "+/ *!!#
M*!+O *#& *!+$
M*%*O "$! #"!
M"*%O && "/0%
M&#!O *0/ "+&*
M+!+O *%% !*+!
M"%!O /$0 *&&$



M'K9O
```

, on the other hand, supports basic regular expressions, so our search expression would look like this:

```
5M $ % ^\/^]O  $ % ^\/^]  $ % ^\&^]
```

We can see that the expression is mostly the same; however, many of the characters that are considered metacharacters in extended expressions are considered literals in basic expressions. They are only treated as metacharacters when escaped with a backslash. De-

pending on the particular configuration of on our system, the matching will be highlighted. If not, try this command mode command:

to activate search highlighting.

---

2 6 Depending on your distribution, may or may not support text search highlighting. Ubuntu, in particular, supplies a very stripped-down version of by default. On such systems, you may want to use your package manager to install a more complete version of .

Summing Up

In this chapter, we've seen a few of the many uses of regular expressions. We can find even more if we use regular expressions to search for additional applications that use them. We can do that by searching the man pages:

```
" - G % 0 % -
```

The B 6 program provides a front end for 6 , allowing it to read compressed files. In our example, we search the compressed section one man page files in their usual location. The result of this command is a list of files containing either the string "regex" or "regular expression". As we can see, regular expressions show up in a lot of programs.

There is one feature found in basic regular expressions that we did not cover. Called ! , this feature will be discussed in the next chapter.

Further Reading

There are many online resources for learning regular expressions, including various tutorials and cheat sheets.

In addition, the Wikipedia has good articles on the following background topics:

POSIX: http://en.wikipedia.org/wiki/Posix

ASCII: http://en.wikipedia.org/wiki/Ascii

0 2

All Unix-like operating systems rely heavily on text files for data storage. So it makes sense that there are many tools for manipulating text. In this chapter, we will look at programs that are used to “slice and dice” text. In the next chapter, we will look at more text processing, focusing on programs that are used to format text for printing and other kinds of human consumption.

This chapter will revisit some old friends and introduce us to some new ones:

– Concatenate files and print on the standard output

– Sort lines of text files

A – Report or omit repeated lines

– Remove sections from each line of files

6 – Merge lines of files

– Join lines of two files on a common field

– Compare two sorted files line by line

– Compare files line by line

6 – Apply a diff file to an original

– Translate or delete characters

– Stream editor for filtering and transforming text

6 – Interactive spell checker

Applications Of Text

So far, we have learned a couple of text editors (and), looked at a bunch of configuration files, and have witnessed the output of dozens of commands, all in text. But what else is text used for? For many things, it turns out.

Documents

Many people write documents using plain text formats. While it is easy to see how a small text file could be useful for keeping simple notes, it is also possible to write large documents in a text format and then embed a to describe the formatting of the finished document. Many scientific papers are written using this method, as Unix-based text processing systems were among the first systems that supported the advanced typographical layout needed by writers in technical disciplines.

Web Pages

The world's most popular type of electronic document is probably the web page. Web pages are text documents that use either , % H, % l or J% H1 ! % l as markup languages to describe the document's visual format.

Email

Email is an intrinsically text-based medium. Even non-text attachments are converted into a text representation for transmission. We can see this for ourselves by downloading an email message and then viewing it in . We will see that the message begins with a that describes the source of the message and the processing it received during its journey, followed by the ! of the message with its content.

Printer Output

On Unix-like systems, output destined for a printer is sent as plain text or, if the page contains graphics, is converted into a text format known as 7 # , which is then sent to a program that generates the graphic dots to be printed.

Program Source Code

Many of the command line programs found on Unix-like systems were created to support system administration and software development, and text processing programs are no exception. Many of them are designed to solve software development problems. The reason text processing is important to software developers is that all software starts out as text. # , the part of the program the programmer actually writes, is always in text format.

Revisiting Some Old Friends

Back in Chapter 6 (Redirection), we learned about some commands that are able to ac-

cept standard input in addition to command line arguments. We only touched on them briefly then, but now we will take a closer look at how they can be used to perform text processing.

The program has a number of interesting options. Many of them are used to help better visualize text content. One example is the 3 option, which is used to display non-printing characters in the text. There are times when we want to know if control characters are embedded in our otherwise visible text. The most common of these are tab characters (as opposed to spaces) and carriage returns, often present as end-of-line characters in MS-DOS-style text files. Another common situation is a file containing lines of text with trailing spaces.

Let's create a test file using as a primitive word processor. To do this, we'll just enter the command (along with specifying a file for redirected output) and type our text, followed by ' to properly end the line, then , to indicate to that we have reached end-of-file. In this example, we enter a leading tab character and follow the line with some trailing spaces:

```
                    &
    ) A    8    6       B,  <
```

Next, we will use with the 3 option to display the text:

```
                    :
`@) A    8    6       B,  <
```

As we can see in the results, the tab character in our text is represented by `@. This is a common notation that means "Control-I" which, as it turns out, is the same as a tab character. We also see that a appears at the true end of the line, indicating that our text contains trailing spaces.

MS-DOS Text Vs. Unix Text

One of the reasons you may want to use to look for non-printing characters in text is to spot hidden carriage returns. Where do hidden carriage returns come from? DOS and Windows! Unix and DOS don't define the end of a line the same way in text files. Unix ends a line with a linefeed character (ASCII 10) while MS-DOS and its derivatives use the sequence carriage return (ASCII 13) and linefeed to terminate each line of text.

There are a several ways to convert files from DOS to Unix format. On many Linux systems, there are programs called * and * , which can convert text files to and from DOS format. However, if you don't have * on your system, don't worry. The process of converting text from DOS to Unix format is very simple; it simply involves the removal of the offending carriage returns. That is easily accomplished by a couple of the programs discussed later in this chapter.

also has options that are used to modify text. The two most prominent are , which numbers lines, and , which suppresses the output of multiple blank lines. We can demonstrate thusly:

```
                    &
) A    8


  6        B,  <

   "      ) A    8
   *
   /        6        B,  <
```

In this example, we create a new version of our < test file, which contains two lines of text separated by two blank lines. After processing by with the options, the extra blank line is removed and the remaining lines are numbered. While this is not much of a process to perform on text, it is a process.

The program sorts the contents of standard input, or one or more files specified on the command line, and sends the results to standard output. Using the same technique that we used with , we can demonstrate processing of standard input directly from the keyboard:

```
&
```

After entering the command, we type the letters “c”, “b”, and “a”, followed once again by to indicate end-of-file. We then view the resulting file and see that the lines now appear in sorted order.

Since can accept multiple files on the command line as arguments, it is possible to multiple files into a single sorted whole. For example, if we had three text files and wanted to combine them into a single sorted file, we could do something like this:

```
&  .  .
```

has several interesting options. Here is a partial list:

! "@)

Option	Long Option	Description
		By default, sorting is performed on the entire line, starting with the first character in the line. This option causes sort to ignore leading spaces in lines and calculates sorting based on the first non-whitespace character on the line.
		Makes sorting case-insensitive.

	Performs sorting based on the numeric evaluation of a string. Using this option allows sorting to be performed on numeric values rather than alphabetic values.
	Sort in reverse order. Results are in descending rather than ascending order.
,L >	Sort based on a key field located from to " rather than the entire line. See discussion below.
	Treat each argument as the name of a presorted file. Merge multiple files into a single sorted result without performing any additional sorting.
6 L	Send sorted output to rather than standard output.
6 L	Define the field-separator character. By default fields are separated by spaces or tabs.

Although most of the options above are pretty self-explanatory, some are not. First, let's look at the option, used for numeric sorting. With this option, it is possible to sort values based on numeric values. We can demonstrate this by sorting the results of the command to determine the largest users of disk space. Normally, the command lists the results of a summary in pathname order:

```
                                0
*#*       5  5   5
%+        5  5   5 6   66
!         5  5   5
"%+       5  5   5
/&&       5  5   5
!         5  5   5
"*&!!     5  5   5  ,
!         5  5   5 6
*"&&$     5  5   5 66
&!        5  5   5 66        ,
```

In this example, we pipe the results into to limit the results to the first ten lines. We can produce a numerically sorted list to show the ten largest consumers of space this way:

```
                                    0        0
#$%%&$   5  5    5         6
*&*++$   5  5    5
"%0#+$   5  5    5
"0%"&&   5  5    5
"&+0+&   5  5    5 , 6
"&&/$&   5  5    5   6
"/#!!$   5  5    5
0+#$!    5  5    5
+!$0*    5  5    5 66
+*!&&    5  5    5
```

By using the options, we produce a reverse numerical sort, with the largest values appearing first in the results. This sort works because the numerical values occur at the beginning of each line. But what if we want to sort a list based on some value found within the line? For example, the results of an :

```
                                  0
   "#*%&!
 8      "              /&!*& *$"+ $& $& $* &*
 8      "             "$"##+ *$$0 "" *0 $+ $!  *6
 8      "              "/$/+ *$"+ $* *0 $! **
 8      "              "$##* *$$0 $! "# "$ /&   6
 8      "               /!$$ *$"+ $& "& $/ #"   6 N      ,
 8      "               0#/+ *$"+ $& "% $$ "%   6 N
 8      "               /#0+ *$"+ $& *% $0 #0    6
 8      "              *$!$! *$"+ $" $/ "! $*     *
 8      "             &!%0$& *$"+ "$ $% "0 $*    6 N
```

Ignoring, for the moment, that can sort its results by size, we could use to sort this list by file size, as well:

```
                                  0           " 0
 8      "             !*/&*"+ *$"+ $& $0 "0 &*      6
 8      "             !***+%* *$"+ $& $0 "0 &*       8
 8      "             /0&+#$! *$"+ $/ $0 */ &#    6 *<&
 8      "             /+#&$*$ *$"+ $! *+ "+ "+ A
 8      "             *%*!0+$ *$"+ $% "$ "& /"
 8      "             *%*!0#+ *$"+ $% "$ "& /"
 8      "             *+$**/+ *$"+ "$ "$ "* #+
```

```
8      "          */$&+!& *$"+ "$ "$ "* #+  6
8      "          **&"!/* *$"+ $& $& $# #+  6
8      "          **$*&0+ *$"+ "$ "$ "* #+
```

Many uses of involve the processing of ! , such as the results of the command above. If we apply database terminology to the table above, we would say that each row is a and that each record consists of multiple , such as the file attributes, link count, filename, file size and so on. is able to process individual fields. In database terms, we are able to specify one or more to use as . In the example above, we specify the and options to perform a reverse numerical sort and specify # to make use the fifth field as the key for sorting.

The option is very interesting and has many features, but first we need to talk about how defines fields. Let's consider a very simple text file consisting of a single line containing the author's name:

```
.          (
```

By default, sees this line as having two fields. The first field contains the characters:

c. d

and the second field contains the characters:

c (d

meaning that whitespace characters (spaces and tabs) are used as delimiters between fields and that the delimiters are included in the field when sorting is performed.

Looking again at a line from our output, we can see that a line contains eight fields and that the fifth field is the file size:

```
8      "          !*/&*"+ *$"+ $& $0 "0 &*      6
```

For our next series of experiments, let's consider the following file containing the history of three popular Linux distributions released from 2006 to 2008. Each line in the file has three fields: the distribution name, version number, and date of release in MM/DD/YYYY format:

```
(2('        "$<*        "*5$05*$$+
            "$          ""5*#5*$$!
(2('        ""<$        $+5"%5*$$!
2           !<$&        $&5*&5*$$!
            !           ""5$!5*$$0
(2('        "$</        "$5$&5*$$0
2           +<"$        "$5*+5*$$+
            0           $#5/"5*$$0
2           0<"$        "$5"!5*$$0
2           0<$&        $&5"%5*$$0
(2('        "$<"        $#5""5*$$+
            +           "$5*&5*$$+
            %           $#5"/5*$$!
2           +<$+        $+5$"5*$$+
2           !<"$        "$5/$5*$$!
            #           $/5*$5*$$+
```

Using a text editor (perhaps), we'll enter this data and name the resulting file < .

Next, we'll try sorting the file and observe the results:

```
            "$          ""5*#5*$$!
            #           $/5*$5*$$+
            +           "$5*&5*$$+
            0           $#5/"5*$$0
            !           ""5$!5*$$0
            %           $#5"/5*$$!
(2('        "$<"        $#5""5*$$+
(2('        "$<*        "*5$05*$$+
(2('        "$</        "$5$&5*$$0
(2('        ""<$        $+5"%5*$$!
2           +<$+        $+5$"5*$$+
2           +<"$        "$5*+5*$$+
2           0<$&        $&5"%5*$$0
2           0<"$        "$5"!5*$$0
2           !<$&        $&5*&5*$$!
2           !<"$        "$5/$5*$$!
```

Well, it mostly worked. The problem occurs in the sorting of the Fedora version numbers. Since a "1" comes before a "5" in the character set, version "10" ends up at the top while version "9" falls to the bottom.

To fix this problem we are going to have to sort on multiple keys. We want to perform an alphabetic sort on the first field and then a numeric sort on the second field. allows

multiple instances of the option so that multiple sort keys can be specified. In fact, a key may include a range of fields. If no range is specified (as has been the case with our previous examples), uses a key that begins with the specified field and extends to the end of the line. Here is the syntax for our multi-key sort:

```
                                !$ ;     !$
            #           $/5*$5*$$+
            +           "$5*&5*$$+
            0           $#5/"5*$$0
            !           ""5$!5*$$0
            %           $#5"/5*$$!
            "$          ""5*#5*$$!
(2('        "$<"        $#5""5*$$+
(2('        "$<*        "*5$05*$$+
(2('        "$</        "$5$&5*$$0
(2('        ""<$        $+5"%5*$$!
2           +<$+        $+5$"5*$$+
2           +<"$        "$5*+5*$$+
2           0<$&        $&5"%5*$$0
2           0<"$        "$5"!5*$$0
2           !<$&        $&5*&5*$$!
2           !<"$        "$5/$5*$$!
```

Though we used the long form of the option for clarity, ">" * would be exactly equivalent. In the first instance of the key option, we specified a range of fields to include in the first key. Since we wanted to limit the sort to just the first field, we specified ">" which means “start at field one and end at field one.” In the second instance, we specified * , which means that field 2 is the sort key and that the sort should be numeric. An option letter may be included at the end of a key specifier to indicate the type of sort to be performed. These option letters are the same as the global options for the program: (ignore leading blanks), (numeric sort), (reverse sort), and so on.

The third field in our list contains a date in an inconvenient format for sorting. On computers, dates are usually formatted in YYYY-MM-DD order to make chronological sorting easy, but ours are in the American format of MM/DD/YYYY. How can we sort this list in chronological order?

Fortunately, provides a way. The key option allows specification of within fields, so we can define keys within fields:

```
                                C           A
            "$          ""5*#5*$$!
2           !<"$        "$5/$5*$$!
```

```
(2('        ""<$        $+5"%5*$$!
            %           $#5"/5*$$!
2           !<$&        $&5*&5*$$!
            !           ""5$!5*$$0
2           0<"$        "$5"!5*$$0
(2('        "$</        "$5$&5*$$0
            0           $#5/"5*$$0
2           0<$&        $&5"%5*$$0
(2('        "$<*        "*5$05*$$+
2           +<"$        "$5*+5*$$+
            +           "$5*&5*$$+
2           +<$+        $+5$"5*$$+
(2('        "$<"        $#5""5*$$+
            #           $/5*$5*$$+
```

By specifying /<0 we instruct to use a sort key that begins at the seventh character within the third field, which corresponds to the start of the year. Likewise, we specify /<" and /<& to isolate the month and day portions of the date. We also add the and options to achieve a reverse numeric sort. The option is included to suppress the leading spaces (whose numbers vary from line to line, thereby affecting the outcome of the sort) in the date field.

Some files don't use tabs and spaces as field delimiters; for example, the 5 56 8 file:

```
   $ $   5  5 5
    " "    5 5  5 5
   * *  5  5 5
,  / / , 5  5 5
,   & +##/& , 5  5 5 ,
    # +$    5 5   5 5
   + "*  5 5  5  5 5
6  0 0 6 5  5 6  5 6 5  5
   ! !   5 5  5 5
 8  % %  8 5  5 6  5 8 5  5
```

The fields in this file are delimited with colons ( ), so how would we sort this file using a key field? provides the option to define the field separator character. To sort the 6 8 file on the seventh field (the account's default shell), we could do this:

```
                        %2%  C        0
  "$$" "$$" -,  >>> 5  5 5 5
```

```
   $ $    5   5  5
  6  "$" "$*  5       5  5
    "$+ ""& ?   9  6  , -    5  5  5   5  5
 6  6  "$& 0 R:=@:  ,      >>> 5  5  5 6  6 5  5
    "$/ "$&  5   5    5  5
       "$! ""%  5  5  5   5  5
6       ""$ "** :   ,1  >>> 5  5  5:   ,1  5  5
6     "$0 ""+ :   3       >>> 5  5  56   5  5
```

By specifying the colon character as the field separator, we can sort on the seventh field.

A

Compared to , the A program is a lightweight. A performs a seemingly trivial task. When given a sorted file (or standard input), it removes any duplicate lines and sends the results to standard output. It is often used in conjunction with to clean the output of duplicates.

(6 While A is a traditional Unix tool often used with , the GNU version of supports a option, which removes duplicates from the sorted output.

Let’s make a text file to try this out:

```
                        &
```

Remember to type to terminate standard input. Now, if we run A on our text file:

```
                        ,
```

the results are no different from our original file; the duplicates were not removed. For A to do its job, the input must be sorted first:

```
                    0  ,
```

This is because A only removes duplicate lines which are adjacent to each other.

A has several options. Here are the common ones:

! "@ " =)

Option	Description
	Output a list of duplicate lines preceded by the number of times the line occurs.
	Only output repeated lines, rather than unique lines.
	Ignore leading fields in each line. Fields are separated by whitespace as they are in ; however, unlike , A has no option for setting an alternate field separator.
	Ignore case during the line comparisons.
	Skip (ignore) the leading characters of each line.
	Only output unique lines. Lines with duplicates are ignored.

Here we see A used to report the number of duplicates found in our text file, using the option:

```
                    0  ,
  *
  *
  *
```

Slicing And Dicing

The next three programs we will discuss are used to peel columns of text out of files and recombine them in useful ways.

The program is used to extract a section of text from a line and output the extracted section to standard output. It can accept multiple file arguments or input from standard input.

Specifying the section of the line to be extracted is somewhat awkward and is specified using the following options:

! "@ (#)

Option	Description
	Extract the portion of the line defined by $. The list may consist of one or more comma-separated numerical ranges.
	Extract one or more fields from the line as defined by $. The list may contain one or more fields or field ranges separated by commas.
	When is specified, use $ as the field delimiting character. By default, fields must be separated by a single tab character.
6	Extract the entire line of text, except for those portions specified by and/or .

As we can see, the way extracts text is rather inflexible. is best used to extract text from files that are produced by other programs, rather than text directly typed by humans. We'll take a look at our < file to see if it is "clean" enough to be a good specimen for our examples. If we use with the 3 option, we can see if the file meets our requirements of tab-separated fields:

```
                    :
(2(`@"$<*`@"*5$05*$$+
    `@"$`@""5*#5*$$!
(2(`@""<$`@$+5"%5*$$!
2   `@!<$&`@$&5*&5*$$!
    `@!`@""5$!5*$$0
(2(`@"$</`@"$5$&5*$$0
2   `@+<"$`@"$5*+5*$$+
    `@0`@$#5/"5*$$0
2   `@0<"$`@"$5"!5*$$0
2   `@0<$&`@$&5"%5*$$0
```

```
(2(`@"$<"`@$#5""5*$$+
    `@+`@"$5*&5*$$+
    `@%`@$#5"/5*$$!
2   `@+<$+`@$+5$"5*$$+
2   `@!<"$`@"$5/$5*$$!
    `@#`@$/5*$5*$$+
```

It looks good. No embedded spaces, just single tab characters between the fields. Since the file uses tabs rather than spaces, we'll use the option to extract a field:

```
"*5$05*$$+
""5*#5*$$!
$+5"%5*$$!
$&5*&5*$$!
""5$!5*$$0
"$5$&5*$$0
"$5*+5*$$+
$#5/"5*$$0
"$5"!5*$$0
$&5"%5*$$0
$#5""5*$$+
"$5*&5*$$+
$#5"/5*$$!
$+5$"5*$$+
"$5/$5*$$!
$/5*$5*$$+
```

Because our file is tab-delimited, it is best to use to extract fields rather than characters. This is because when a file is tab-delimited, it is unlikely that each line will contain the same number of characters, which makes calculating character positions within the line difficult or impossible. In our example above, however, we now have extracted a field that luckily contains data of identical length, so we can show how character extraction works by extracting the year from each line:

```
                                0       C *
*$$+
*$$!
*$$!
*$$!
*$$0
*$$0
*$$+
*$$0
```

```
*$$0
*$$0
*$$+
*$$+
*$$!
*$$+
*$$!
*$$+
```

By running a second time on our list, we are able to extract character positions 7 through 10, which corresponds to the year in our date field. The 0 "$ notation is an example of a range. The man page contains a complete description of how ranges can be specified.

Expanding Tabs

Our < file is ideally formatted for extracting fields using . But what if we wanted a file that could be fully manipulated with by characters, rather than fields? This would require us to replace the tab characters within the file with the corresponding number of spaces. Fortunately, the GNU Coreutils package includes a tool for that. Named 6 , this program accepts either one or more file arguments or standard input, and outputs the modified text to standard output.

If we process our < file with 6 , we can use the to extract any range of characters from the file. For example, we could use the following command to extract the year of release from our list, by expanding the file and using to extract every character from the twenty-third position to the end of the line:

```
0
```

Coreutils also provides the 6 program to substitute tabs for spaces.

When working with fields, it is possible to specify a different field delimiter rather than the tab character. Here we will extract the first field from the 5 56 8 file:

```
%2%  0
```

```
,
'


6

 8
```

Using the option, we are able to specify the colon character as the field delimiter.

6

The 6 command does the opposite of . Rather than extracting a column of text from a file, it adds one or more columns of text to a file. It does this by reading multiple files and combining the fields found in each file into a single stream on standard output. Like , 6 accepts multiple file arguments and/or standard input. To demonstrate how 6 operates, we will perform some surgery on our < file to produce a chronological list of releases.

From our earlier work with , we will first produce a list of distros sorted by date and store the result in a file called , < :

```
                          C          A          &
   !
```

Next, we will use to extract the first two fields from the file (the distro name and version), and store that result in a file named < :

```
                          ;       !       &


          "$
2         !<"$
(2('      ""<$
          %
2         !<$&
          !
2         0<"$
(2('      "$</
          0
2         0<$&
```

The final piece of preparation is to extract the release dates and store them a file named < :

```
                                    !       &

""5*#5*$$!
"$5/$5*$$!
$+5"%5*$$!
$#5"/5*$$!
$&5*&5*$$!
""5$!5*$$0
"$5"!5*$$0
"$5$&5*$$0
$#5/"5*$$0
$&5"%5*$$0
```

We now have the parts we need. To complete the process, use 6 to put the column of dates ahead of the distro names and versions, thus creating a chronological list. This is done simply by using 6 and ordering its arguments in the desired arrangement:

```
""5*#5*$$!                  "$
"$5/$5*$$!          2       !<"$
$+5"%5*$$!          (2('    ""<$
$#5"/5*$$!                  %
$&5*&5*$$!          2       !<$&
""5$!5*$$0                  !
"$5"!5*$$0          2       0<"$
"$5$&5*$$0          (2('    "$</
$#5/"5*$$0                  0
$&5"%5*$$0          2       0<$&
"*5$05*$$+          (2('    "$<*
"$5*+5*$$+          2       +<"$
"$5*&5*$$+                  +
$+5$"5*$$+          2       +<$+
$#5""5*$$+          (2('    "$<"
$/5*$5*$$+                  #
```

In some ways, is like 6 in that it adds columns to a file, but it uses a unique way to do it. A A is an operation usually associated with ! where data from multiple ! with a shared key field is combined to form a desired result.

The program performs the same operation. It joins data from multiple files based on a shared key field.

To see how a join operation is used in a relational database, let's imagine a very small database consisting of two tables, each containing a single record. The first table, called CUSTOMERS, has three fields: a customer number (CUSTNUM), the customer's first name (FNAME), and the customer's last name (LNAME):

```
 2()K2-        K3-'          =K3-'
LLLLLLLL      LLLLL         LLLLLL
&+!"%/&                     (
```

The second table is called ORDERS and contains four fields: an order number (ORDERNUM), the customer number (CUSTNUM), the quantity (QUAN), and the item ordered (ITEM).

```
SQ9'QK2-          2()K2-        [23K   @)'-
LLLLLLLL          LLLLLLL       LLLL   LLLL
/$"&%#//$#        &+!"%/&       "      G    .
```

Note that both tables share the field CUSTNUM. This is important, as it allows a relationship between the tables.

Performing a join operation would allow us to combine the fields in the two tables to achieve a useful result, such as preparing an invoice. Using the matching values in the CUSTNUM fields of both tables, a join operation could produce the following:

```
 K3-'          =K3-'         [23K   @)'-
LLLLL         LLLLL         LLLL   LLLL
              (             "      G    .
```

To demonstrate the program, we'll need to make a couple of files with a shared key. To do this, we will use our , < file. From this file, we will construct two additional files, one containing the release dates (which will be our shared key for this demonstration) and the release names:

```
                                ;       !       &
                                                &
  !
                                    !
""5*#5*$$!
"$5/$5*$$!        2
$+5"%5*$$!        (2('
$#5"/5*$$!
$&5*&5*$$!        2
""5$!5*$$0
"$5"!5*$$0        2
```

```
"$5$&5*$$0        (2('
$#5/"5*$$0
$&5"%5*$$0        2
```

and the second file, which contains the release dates and the version numbers:

```
                                ;        !        &
                                                        &
   !
                                    !
""5*#5*$$!          "$
"$5/$5*$$!          !<"$
$+5"%5*$$!          ""<$
$#5"/5*$$!          %
$&5*&5*$$!          !<$&
""5$!5*$$0          !
"$5"!5*$$0          0<"$
"$5$&5*$$0          "$</
$#5/"5*$$0          0
$&5"%5*$$0          0<$&
```

We now have two files with a shared key (the “release date” field). It is important to point out that the files must be sorted on the key field for to work properly.

```
                                    !                !          0

""5*#5*$$!        "$
"$5/$5*$$! 2      !<"$
$+5"%5*$$! (2(' ""<$
$#5"/5*$$!        %
$&5*&5*$$! 2      !<$&
""5$!5*$$0        !
"$5"!5*$$0 2      0<"$
"$5$&5*$$0 (2(' "$</
$#5/"5*$$0        0
$&5"%5*$$0 2      0<$&
```

Note also that, by default, uses whitespace as the input field delimiter and a single space as the output field delimiter. This behavior can be modified by specifying options. See the man page for details.

Comparing Text

It is often useful to compare versions of text files. For system administrators and software developers, this is particularly important. A system administrator may, for example, need to compare an existing configuration file to a previous version to diagnose a system problem. Likewise, a programmer frequently needs to see what changes have been made to programs over time.

The program compares two text files and displays the lines that are unique to each one and the lines they have in common. To demonstrate, we will create two nearly identical text files using :

```
&

&
```

Next, we will compare the two files using :

```

```

As we can see, produces three columns of output. The first column contains lines unique to the first file argument; the second column, the lines unique to the second file argument; the third column contains the lines shared by both files. supports options in the form where is either 1, 2 or 3. When used, these options specify which column(s) to suppress. For example, if we only wanted to output the lines shared by both files, we would suppress the output of columns one and two:

```

```

Like the program, is used to detect the differences between files. However, is a much more complex tool, supporting many output formats and the ability to process large collections of text files at once. is often used by software developers to examine changes between different versions of program source code, and thus has the ability to recursively examine directories of source code, often referred to as . One common use for is the creation of or that are used by programs such as 6 (which we'll discuss shortly) to convert one version of a file (or files) to another version.

If we use to look at our previous example files:

```
" $
X
& &
C
```

we see its default style of output: a terse description of the differences between the two files. In the default format, each group of changes is preceded by a in the form of to describe the positions and types of changes required to convert the first file to the second file:

! "@ -

Change	Description
	Append the lines at the position " in the second file to the position in the first file.
	Change (replace) the lines at position with the lines at the position " in the second file.
	Delete the lines in the first file at position , which would have appeared at range " in the second file

In this format, a range is a comma-separated list of the starting line and the ending line. While this format is the default (mostly for POSIX compliance and backward compatibility with traditional Unix versions of), it is not as widely used as other, optional formats. Two of the more popular formats are the and the .

When viewed using the context format (the option), we will see this:

```
DDD     "<          *$$! "* */ $+ && "/<$$$$$$$$$  $#$$
       *<           *$$! "* */ $+ && /&<$$$$$$$$$  $#$$
DDDDDDDDDDDDDDD
DDD ">& DDDD




   ">&



7
```

The output begins with the names of the two files and their timestamps. The first file is marked with asterisks and the second file is marked with dashes. Throughout the remainder of the listing, these markers will signify their respective files. Next, we see groups of changes, including the default number of surrounding context lines. In the first group, we see:

DDD ">& DDD

which indicates lines 1 through 4 in the first file. Later we see:

   ">&

which indicates lines 1 through 4 in the second file. Within a change group, lines begin with one of four indicators:

! "@ 2 +

Indicator	Meaning
	A line shown for context. It does not indicate a difference between the two files.
	A line deleted. This line will appear in the first file but not in the second file.

7	A line added. This line will appear in the second file but not in the first file.
F	A line changed. The two versions of the line will be displayed, each in its respective section of the change group.

The unified format is similar to the context format but is more concise. It is specified with the option:

```
     "<            *$$! "* */ $+ && "/<$$$$$$$$$  $#$$
777    *<          *$$! "* */ $+ && /&<$$$$$$$$$  $#$$
   ">& 7">&




7
```

The most notable difference between the context and unified formats is the elimination of the duplicated lines of context, making the results of the unified format shorter than those of the context format. In our example above, we see file timestamps like those of the context format, followed by the string ">& 7">& . This indicates the lines in the first file and the lines in the second file described in the change group. Following this are the lines themselves, with the default three lines of context. Each line starts with one of three possible characters:

! "@ 3 0 +

Character	Meaning
	This line is shared by both files.
	This line was removed from the first file.
7	This line was added to the first file.

6

The 6 program is used to apply changes to text files. It accepts output from and is generally used to convert older version of files into newer versions. Let’s consider a famous example. The Linux kernel is developed by a large, loosely organized team of

contributors who submit a constant stream of small changes to the source code. The Linux kernel consists of several million lines of code, while the changes that are made by one contributor at one time are quite small. It makes no sense for a contributor to send each developer an entire kernel source tree each time a small change is made. Instead, a diff file is submitted. The diff file contains the change from the previous version of the kernel to the new version with the contributor's changes. The receiver then uses the 6 program to apply the change to his own source tree. Using /6 offers two significant advantages:

1. The diff file is very small, compared to the full size of the source tree.
2. The diff file concisely shows the change being made, allowing reviewers of the patch to quickly evaluate it.

Of course, /6 will work on any text file, not just source code. It would be equally applicable to configuration files or any other text.

To prepare a diff file for use with 6 , the GNU documentation (see Further Reading below) suggests using as follows:

K C

Where $ and $ are either single files or directories containing files. The option supports recursion of a directory tree.

Once the diff file has been created, we can apply it to patch the old file into the new file:

6 X

We'll demonstrate with our test file:

```
                    ?          &
                    /
6          "<
```

In this example, we created a diff file named 6 < and then used the 6 program to apply the patch. Note that we did not have to specify a target file to 6 , as the diff file (in unified format) already contains the filenames in the header. Once the patch is applied, we can see that "< now matches *< .

6 has a large number of options, and there are additional utility programs that can be used to analyze and edit patches.

Editing On The Fly

Our experience with text editors has been largely , meaning that we manually move a cursor around, then type our changes. However, there are ways to edit text as well. It's possible, for example, to apply a set of changes to multiple files with a single command.

The program is used to characters. We can think of this as a sort of character-based search-and-replace operation. Transliteration is the process of changing characters from one alphabet to another. For example, converting characters from lowercase to uppercase is transliteration. We can perform such a conversion with as follows:

```
                    J          J 0    - : @
=S.'Q 3(' =))'Q(
```

As we can see, operates on standard input, and outputs its results on standard output. accepts two arguments: a set of characters to convert from and a corresponding set of characters to convert to. Character sets may be expressed in one of three ways:

1. An enumerated list. For example, 3G 9' ?R@ 1=-KS:[Q()2;.VUH
2. A character range. For example, 3 H . Note that this method is sometimes subject to the same issues as other commands, due to the locale collation order, and thus should be used with caution.
3. POSIX character classes. For example, 66 .

In most cases, both character sets should be of equal length; however, it is possible for the first set to be larger than the second, particularly if we wish to convert multiple characters to a single character:

```
                    J          J 0   (2    2) :
333333333 3333333
```

In addition to transliteration, allows characters to simply be deleted from the input stream. Earlier in this chapter, we discussed the problem of converting MS-DOS text files to Unix-style text. To perform this conversion, carriage return characters need to be removed from the end of each line. This can be performed with as follows:

J^ J X C

where $ is the file to be converted and $ is the result. This form of the command uses the escape sequence ^ to represent the carriage return character. To see a complete list of the sequences and character classes supports, try:

```
```

ROT13: The Not-So-Secret Decoder Ring

One amusing use of is to perform :) (of text. ROT13 is a trivial type of encryption based on a simple substitution cipher. Calling ROT13 “encryption” is being generous; “text obfuscation” is more accurate. It is used sometimes on text to obscure potentially offensive content. The method simply moves each character 13 places up the alphabet. Since this is half way up the possible 26 characters, performing the algorithm a second time on the text restores it to its original form. To perform this encoding with :

```
J     J0   -:@  -  ?@:O
6
```

Performing the same procedure a second time results in the translation:

```
J     J0   -:@  -  ?@:O
```

A number of email programs and Usenet news readers support ROT13 encoding. Wikipedia contains a good article on the subject:

http://en.wikipedia.org/wiki/ROT13

can perform another trick, too. Using the option, can “squeeze” (delete) repeated instances of a character:

```
                J    J0
```

Here we have a string containing repeated characters. By specifying the set “ab” to , we eliminate the repeated instances of the letters in the set, while leaving the character that is missing from the set (“c”) unchanged. Note that the repeating characters must be adjoining. If they are not:

```
J   JO
```

the squeezing will have no effect.

The name is short for . It performs text editing on a stream of text, either a set of specified files or standard input. is a powerful and somewhat complex program (there are entire books about it), so we will not cover it completely here.

In general, the way works is that it is given either a single editing command (on the command line) or the name of a script file containing multiple commands, and it then performs these commands upon each line in the stream of text. Here is a very simple example of in action:

```
J  JO  %     %
```

In this example, we produce a one-word stream of text using echo and pipe it into . , in turn, carries out the instruction 5 5 5 upon the text in the stream and produces the output “back” as a result. We can also recognize this command as resembling the “substitution” (search-and-replace) command in .

Commands in begin with a single letter. In the example above, the substitution command is represented by the letter and is followed by the search-and-replace strings, separated by the slash character as a delimiter. The choice of the delimiter character is arbitrary. By convention, the slash character is often used, but will accept any character that immediately follows the command as the delimiter. We could perform the same command this way:

```
J  JO  %.  .  .%
```

By using the underscore character immediately after the command, it becomes the delimiter. The ability to set the delimiter can be used to make commands more readable, as we shall see.

Most commands in may be preceded by an , which specifies which line(s) of the input stream will be edited. If the address is omitted, then the editing command is car-

ried out on every line in the input stream. The simplest form of address is a line number. We can add one to our example:

```
                J  J0  %      %
```

Adding the address " to our command causes our substitution to be performed on the first line of our one-line input stream. If we specify another number:

```
                J  J0  %      %
```

we see that the editing is not carried out, since our input stream does not have a line 2.

Addresses may be expressed in many ways. Here are the most common:

! "@ 4 * >

Address	Description
	A line number where is a positive integer.
	The last line.
5 5	Lines matching a POSIX basic regular expression. Note that the regular expression is delimited by slash characters. Optionally, the regular expression may be delimited by an alternate character, by specifying the expression with ^ , where is the alternate character.
>	A range of lines from to " , inclusive. Addresses may be any of the single address forms above.
	Match the line represented by the number , then each subsequent line at intervals. For example 1~2 refers to each odd numbered line, 5~5 refers to the fifth line and every fifth line thereafter.
>7	Match and the following lines.
!	Match all lines except , which may be any of the forms above.

We’ll demonstrate different kinds of addresses using the < file from earlier

in this chapter. First, a range of line numbers:

```
                                % ;" %
(2('        "$<*          "*5$05*$$+
            "$            ""5*#5*$$!
(2('        ""<$          $+5"%5*$$!
2           !<$&          $&5*&5*$$!
            !       ""5$!5*$$0
```

In this example, we print a range of lines, starting with line 1 and continuing to line 5. To do this, we use the 6 command, which simply causes a matched line to be printed. For this to be effective however, we must include the option (the no auto-print option) to cause not to print every line by default.

Next, we'll try a regular expression:

```
                                % FEFG  %
(2('        "$<*          "*5$05*$$+
(2('        ""<$          $+5"%5*$$!
(2('        "$</          "$5$&5*$$0
(2('        "$<"          $#5""5*$$+
```

By including the slash-delimited regular expression 5(2('5 , we are able to isolate the lines containing it in much the same manner as 6 .

Finally, we'll try negation by adding an exclamation point (F) to the address:

```
                                % FEFG L %
            "$            ""5*#5*$$!
2           !<$&          $&5*&5*$$!
            !             ""5$!5*$$0
2           +<"$          "$5*+5*$$+
            0             $#5/"5*$$0
2           0<"$          "$5"!5*$$0
2           0<$&          $&5"%5*$$0
            +             "$5*&5*$$+
            %             $#5"/5*$$!
2           +<$+          $+5$"5*$$+
2           !<"$          "$5/$5*$$!
            #             $/5*$5*$$+
```

Here we see the expected result: all of the lines in the file except the ones matched by the

regular expression.

So far, we've looked at two of the editing commands, and 6. Here is a more complete list of the basic editing commands:

! "@ 5 9 1

Command	Description
L	Output current line number.
	Append text after the current line.
	Delete the current line.
	Insert text in front of the current line.
6	Print the current line. By default, prints every line and only edits lines that match a specified address within the file. The default behavior can be overridden by specifying the option.
A	Exit without processing any more lines. If the option is not specified, output the current line.
[	Exit without processing any more lines.
5 5 5	Substitute the contents of wherever is found. may include the special character Y, which is equivalent to the text matched by . In addition, may include the sequences ^" through ^%, which are the contents of the corresponding subexpressions in . For more about this, see the discussion of ! below. After the trailing slash following , an optional flag may be specified to modify the command's behavior.
,5 5	Perform transliteration by converting characters from to the corresponding characters in " . Note that unlike , requires that both sets be of the same length.

The command is by far the most commonly used editing command. We will demonstrate just some of its power by performing an edit on our < file. We discussed before how the date field in < was not in a "computer-friendly" for-

mat. While the date is formatted MM/DD/YYYY, it would be better (for ease of sorting) if the format were YYYY-MM-DD. To perform this change on the file by hand would be both time consuming and error prone, but with , this change can be performed in one step:

```
                                %  K4(* D)K9 K>K6K K4(* D)K9 K>K6K K4(* D)K9AK>K
63 K  K  K  %
(2('          "$<*          *$$+ "* $0
              "$            *$$! "" *#
(2('          ""<$          *$$! $+ "%
2             !<$&          *$$! $& *&
              !             *$$0 "" $!
(2('          "$</          *$$0 "$ $&
2             +<"$          *$$+ "$ *+
              0             *$$0 $# /"
2             0<"$          *$$0 "$ "!
2             0<$&          *$$0 $& "%
(2('          "$<"          *$$+ $# ""
              +             *$$+ "$ *&
              %             *$$! $# "/
2             +<$+          *$$+ $+ $"
2             !<"$          *$$! "$ /$
              #             *$$+ $/ *$
```

Wow! Now that is an ugly looking command. But it works. In just one step, we have changed the date format in our file. It is also a perfect example of why regular expressions are sometimes jokingly referred to as a “write-only” medium. We can write them, but we sometimes cannot read them. Before we are tempted to run away in terror from this command, let’s look at how it was constructed. First, we know that the command will have this basic structure:

```
  %                                %
```

Our next step is to figure out a regular expression that will isolate the date. Since it is in MM/DD/YYYY format and appears at the end of the line, we can use an expression like this:

```
(* D)9 > (* D)9 > (* D)9A>3
```

which matches two digits, a slash, two digits, a slash, four digits, and the end of line. So that takes care of , but what about ? To handle that, we must introduce

a new regular expression feature that appears in some applications which use BRE. This feature is called ! and works like this: If the sequence \ appears in where is a number from 1 to 9, the sequence will refer to the corresponding subexpression in the preceding regular expression. To create the subexpressions, we simply enclose them in parentheses like so:

```
4(* D)9 >6 4(* D)9 >6 4(* D)9A>63
```

We now have three subexpressions. The first contains the month, the second contains the day of the month, and the third contains the year. Now we can construct as follows:

```
K  K  K
```

which gives us the year, a dash, the month, a dash, and the day.

Now, our command looks like this:

```
%  4(* D)9 >6 4(* D)9 >6 4(* D)9A>63 K  K  K  %
```

We have two remaining problems. The first is that the extra slashes in our regular expression will confuse when it tries to interpret the command. The second is that since , by default, accepts only basic regular expressions, several of the characters in our regular expression will be taken as literals, rather than as metacharacters. We can solve both these problems with a liberal application of backslashes to escape the offending characters:

```
%  K4(* D)K9 K>K6K K4(* D)K9 K>K6K K4(* D)K9AK>K63 K  K  K  %
```

And there you have it!

Another feature of the command is the use of optional flags that may follow the replacement string. The most important of these is the flag, which instructs to apply the search-and-replace globally to a line, not just to the first instance, which is the default. Here is an example:

```
                    J     J 0   %   < %
 G
```

We see that the replacement was performed, but only to the first instance of the letter “b,” while the remaining instances were left unchanged. By adding the flag, we are able to change all the instances:

```
                    J     J 0   %   < %
 GGG
```

So far, we have only given single commands via the command line. It is also possible to construct more complex commands in a script file using the option. To demonstrate, we will use with our < file to build a report. Our report will feature a title at the top, our modified dates, and all the distribution names converted to uppercase. To do this, we will need to write a script, so we’ll fire up our text editor and enter the following:

```
U              T

  K
K
T   1        H    K

  K4(* D)K9 K>K6K K4(* D)K9 K>K6K K4(* D)K9AK>K63 K  K  K
!            ,      !- :<=1G7QRPYXTO?SB\HF+E]WZ^@
```

We will save our script as < and run it like this:

```
=    9         Q 6

(2('         "$<*         *$$+ "* $0
 '9SQ3       "$           *$$! "" *#
(2('         ""<$         *$$! $+ "%
2G2K)2       !<$&         *$$! $& *&
 '9SQ3       !            *$$0 "" $!
(2('         "$</         *$$0 "$ $&
2G2K)2       +<"$         *$$+ "$ *+
 '9SQ3       0            *$$0 $# /"
2G2K)2       0<"$         *$$0 "$ "!
```

```
2G2K)2    0<$&        *$$0 $& "%
(2('      "$<"        *$$+ $# ""
 '9SQ3    +           *$$+ "$ *&
 '9SQ3    %           *$$! $# "/
2G2K)2    +<$+        *$$+ $+ $"
2G2K)2    !<"$        *$$! "$ /$
 '9SQ3    #           *$$+ $/ *$
```

As we can see, our script produces the desired results, but how does it do it? Let's take another look at our script. We'll use to number the lines:

```
"        Z      6    6     =              6
*
/        "  ^
&        ^
#        =    9          Q 6   ^
+
0         5^M $ % ^\*^]^O^5^M $ % ^\*^]^O^5^M $ % ^\&^]^O 5^/ ^" ^*5
!        ,5            6A    8 ,B53G 9' ?R@ 1=-KS:[Q()2;.VUH5
```

Line one of our script is a . Like many configuration files and programming languages on Linux systems, comments begin with the Z character and are followed by human-readable text. Comments can be placed anywhere in the script (though not within commands themselves) and are helpful to any humans who might need to identify and/or maintain the script.

Line 2 is a blank line. Like comments, blank lines may be added to improve readability.

Many commands support line addresses. These are used to specify which lines of the input are to be acted upon. Line addresses may be expressed as single line numbers, line number ranges, and the special line number "$" which indicates the last line of input.

Lines 3 through 6 contain text to be inserted at the address 1, the first line of the input. The command is followed by the sequence backslash-carriage return to produce an escaped carriage return, or what is called a . This sequence, which can be used in many circumstances including shell scripts, allows a carriage return to be embedded in a stream of text without signaling the interpreter (in this case) that the end of the line has been reached. The , and likewise, the (which appends text, rather than inserting it) and (which replaces text) commands, allow multiple lines of text as long as each line, except the last, ends with a line-continuation character. The sixth line of our script is actually the end of our inserted text and ends with a plain carriage return rather than a line-continuation character, signaling the end of the command.

2 6 A line-continuation character is formed by a backslash followed by a carriage return. No intermediary spaces are permitted.

Line 7 is our search-and-replace command. Since it is not preceded by an address, each line in the input stream is subject to its action.

Line 8 performs transliteration of the lowercase letters into uppercase letters. Note that unlike , the , command in does not support character ranges (for example, B), nor does it support POSIX character classes. Again, since the , command is not preceded by an address, it applies to every line in the input stream.

People Who Like Also Like...

is a very capable program, able to perform fairly complex editing tasks to streams of text. It is most often used for simple, one-line tasks rather than long scripts. Many users prefer other tools for larger tasks. The most popular of these are 8 and 6 . These go beyond mere tools like the programs covered here, and extend into the realm of complete programming languages. 6 , in particular, is often used instead of shell scripts for many system-management and administration tasks, as well as being a very popular medium for web development. 8 is a little more specialized. Its specific strength is its ability to manipulate tabular data. It resembles in that 8 programs normally process text files line-by-line, using a scheme similar to the concept of an address followed by an action. While both 8 and 6 are outside the scope of this book, they are very good skills for the Linux command line user to learn.

6

The last tool we will look at is 6 , an interactive spelling checker. The 6 program is the successor to an earlier program named 6 , and can be used, for the most part, as a drop-in replacement. While the 6 program is mostly used by other programs that require spell-checking capability, it can also be used very effectively as a stand-alone tool from the command line. It has the ability to intelligently check various types of text files, including HTML documents, C/C++ programs, email messages, and other kinds of specialized texts.

To spell check a text file containing simple prose, it could be used like this:

```
 6
```

where is the name of the file to check. As a practical example, let's create a simple text file named < containing some deliberate spelling errors:

```
                    &
+  ,                !
```

Next we'll check the file using 6 :

```

```

As 6 is interactive in the check mode, we will see a screen like this:

```
)  A     8                 6          ,   <

[redacted]
"O   6                    +O 8  6
*O   6                    0O   6
/O   6                    !O   6
&O   6                     %O   6
#O 6  6                     $O   6
 O @                        @O @
 O Q 6                      QO Q 6
 O 3                        O 3   = 8
 O 3                        O '
[redacted]
E
```

At the top of the display, we see our text with a suspiciously spelled word highlighted. In the middle, we see ten spelling suggestions numbered zero through nine, followed by a list of other possible actions. Finally, at the very bottom, we see a prompt ready to accept our choice.

If we press the " key, 6 replaces the offending word with the word "jumped" and moves on to the next misspelled word, which is "laxy." If we select the replacement "lazy," 6 replaces it and terminates. Once 6 has finished, we can examine our file and see that the misspellings have been corrected:

```
) A 8 6 B, <
```

Unless told otherwise via the command line option 6 , 6 creates a backup file containing the original text by appending the extension < to the filename.

Showing off our editing prowess, we'll put our spelling mistakes back in so we can reuse our file:

```
% -! ! # %
```

The option tells to edit the file "in-place," meaning that rather than sending the edited output to standard output, it will rewrite the file with the changes applied. We also see the ability to place more than one editing command on the line by separating them with a semicolon.

Next, we'll look at how 6 can handle different kinds of text files. Using a text editor such as (the adventurous may want to try), we will add some HTML markup to our file:

```
/ &
  / &
    / &O R+OT / &
  / &
  / !&
    / & ) A 8 6 , < / &
  / !&
/ &
```

Now, if we try to spell check our modified file, we run into a problem. If we do it this way:

```

```

we'll get this:

```
X[redacted]C
```

```
      X   C
           X    C- 6     R)-=    X5    C
      X5   C
      X  ,C
           X6C)  A      8       6          ,   <X56C
      X5  ,C
X5   C

"O R)-=                          &O R
*O                              #O R
/O                              +O


 O @                             @O @
 O Q 6                           QO Q 6
 O 3                             O 3  = 8
 O 3                             O '
E
```

6 will see the contents of the HTML tags as misspelled. This problem can be overcome by including the R (HTML) checking-mode option, like this:

```
                                R
```

which will result in this:

```
X   C
      X   C
           X    C                - 6          R)-=    X5    C
      X5   C
      X  ,C
           X6C)  A      8       6          ,   <X56C
      X5  ,C
X5   C

"O -  6                         +O -  66
*O -  6                         0O -
/O -  6                         !O Q  6
&O 9  6                          %O -  6
#O (6                           $O -
 O @                             @O @
 O Q 6                           QO Q 6
```

```
O 3                    O 3  = 8
O 3                    O '
[redacted]
E
```

The HTML is ignored and only the non-markup portions of the file are checked. In this mode, the contents of HTML tags are ignored and not checked for spelling. However, the contents of 3=) tags, which benefit from checking, are checked in this mode.

---

2 6 By default, 6 will ignore URLs and email addresses in text. This behavior can be overridden with command line options. It is also possible to specify which markup tags are checked and skipped. See the 6 man page for details.

Summing Up

In this chapter, we have looked at a few of the many command line tools that operate on text. In the next chapter, we will look at several more. Admittedly, it may not seem immediately obvious how or why you might use some of these tools on a day-to-day basis, though we have tried to show some semi-practical examples of their use. We will find in later chapters that these tools form the basis of a tool set that is used to solve a host of practical problems. This will be particularly true when we get into shell scripting, where these tools will really show their worth.

Further Reading

The GNU Project website contains many online guides to the tools discussed in this chapter.

From the Coreutils package:
http://www.gnu.org/software/coreutils/manual/coreutils.html#Output-of-entire-files
http://www.gnu.org/software/coreutils/manual/coreutils.html#Operating-on-sorted-files
http://www.gnu.org/software/coreutils/manual/coreutils.html#Operating-on-fields
http://www.gnu.org/software/coreutils/manual/coreutils.html#Operating-on-characters

From the Diffutils package:
http://www.gnu.org/software/diffutils/manual/html_mono/diff.html

:
http://www.gnu.org/software/sed/manual/sed.html

6 :
http://aspell.net/man-html/index.html

There are many other online resources for , in particular:
http://www.grymoire.com/Unix/Sed.html
http://sed.sourceforge.net/sed1line.txt

Also try googling “sed one liners”, “sed cheat sheets”

Extra Credit

There are a few more interesting text-manipulation commands worth investigating. Among these are: 6 (split files into pieces), 6 (split files into pieces based on context), and (side-by-side merge of file differences).

' $ 7 "

In this chapter, we continue our look at text-related tools, focusing on programs that are used to format text output, rather than changing the text itself. These tools are often used to prepare text for eventual printing, a subject that we will cover in the next chapter. The programs that we will cover in this chapter include:

– Number lines

– Wrap each line to a specified length

– A simple text formatter

6 – Prepare text for printing

6 – Format and print data

– A document formatting system

Simple Formatting Tools

We'll look at some of the simple formatting tools first. These are mostly single-purpose programs, and a bit unsophisticated in what they do, but they can be used for small tasks and as parts of pipelines and scripts.

– Number Lines

The program is a rather arcane tool used to perform a simple task. It numbers lines. In its simplest use, it resembles :

```
                    0
"       (2('        "$<*        "*5$05*$$+
*                   "$          ""5*#5*$$!
/       (2('        ""<$        $+5"%5*$$!
&       2           !<$&        $&5*&5*$$!
#                   !           ""5$!5*$$0
+       (2('        "$</        "$5$&5*$$0
0       2           +<"$        "$5*+5*$$+
```

```
  !                 0           $#5/"5*$$0
  %       2         0<"$        "$5"!5*$$0
 "$       2         0<$&        $&5"%5*$$0
```

Like , can accept either multiple files as command line arguments, or standard input. However, has a number of options and supports a primitive form of markup to allow more complex kinds of numbering.

supports a concept called “logical pages” when numbering. This allows to reset (start over) the numerical sequence when numbering. Using options, it is possible to set the starting number to a specific value and, to a limited extent, its format. A logical page is further broken down into a header, body, and footer. Within each of these sections, line numbering may be reset and/or be assigned a different style. If is given multiple files, it treats them as a single stream of text. Sections in the text stream are indicated by the presence of some rather odd-looking markup added to the text:

! " %

Markup	Meaning
^ ^ ^	Start of logical page header
^ ^	Start of logical page body
^	Start of logical page footer

Each of the above markup elements must appear alone on its own line. After processing a markup element, deletes it from the text stream.

Here are the common options for :

! " ")

Option	Meaning
	Set body numbering to , where is one of the following: = number all lines = number only non-blank lines. This is the default. = none 6 = number only lines matching basic regular expression .
	Set footer numbering to . Default is n (none).
	Set header numbering to . Default is n (none).

	Set page numbering increment to ! . Default is one.
	Sets numbering format to , where format is: = left justified, without leading zeros. = right justified, without leading zeros. This is the default. B = right justified, with leading zeros.
6	Do not reset page numbering at the beginning of each logical page.
	Add to the end of each line number to create a separator. Default is a single tab character.
	Set first line number of each logical page to ! . Default is one.
8	Set width of the line number field to . Default is 6.

Admittedly, we probably won't be numbering lines that often, but we can use to look at how we can combine multiple tools to perform more complex tasks. We will build on our work in the previous chapter to produce a Linux distributions report. Since we will be using , it will be useful to include its header/body/footer markup. To do this, we will add it to the script from the last chapter. Using our text editor, we will change the script as follows and save it as < :

```
Z      6   6    =            6

" ^
KK2KK2KK2K
^
=   9         Q 6  ^
K
?               ]      H     K
                            K
KK2KK2
 5^M $ % ^\*^]^O^5^M $ % ^\*^]^O^5^M $ % ^\&^]^O 5^/ ^" ^*5
3  K
KK2K
K
G   S  H
```

The script now inserts the logical page markup and adds a footer at the end of the report. Note that we had to double up the backslashes in our markup, because they are normally interpreted as an escape character by .

Next, we'll produce our enhanced report by combining , , and :

```
                                ;                0
0


   =    9          Q 6

   K       ; <   Q


 "         #      *$$+ $/ *$
 *         +      *$$+ "$ *&
 /         0      *$$0 $# /"
 &         !      *$$0 "" $!
 #         %      *$$! $# "/
 +         "$     *$$! "" *#
 0  (2('   "$<"   *$$+ $# ""
 !  (2('   "$<*   *$$+ "* $0
 %  (2('   "$</   *$$0 "$ $&
"$  (2('   ""<$   *$$! $+ "%
""  2      +<$+   *$$+ $+ $"
"*  2      +<"$   *$$+ "$ *+
"/  2      0<$&   *$$0 $& "%
"&  2      0<"$   *$$0 "$ "!
"#  2      !<$&   *$$! $& *&
"+  2      !<"$   *$$! "$ /$



   '  S  Q 6
```

Our report is the result of our pipeline of commands. First, we sort the list by distribution name and version (fields 1 and 2), then we process the results with , adding the report header (including the logical page markup for) and footer. Finally, we process the result with , which, by default, only numbers the lines of the text stream that belong to the body section of the logical page.

We can repeat the command and experiment with different options for . Some interesting ones are:

```
   -
```

and

```
        % %
```

– Wrap Each Line To A Specified Length

is the process of breaking lines of text at a specified width. Like our other commands, accepts either one or more text files or standard input. If we send a simple stream of text, we can see how it works:

```
                    J+  ,                    -!   J
0
)  A
 8      6

 B,   <
```

Here we see in action. The text sent by the command is broken into segments specified by the 8 option. In this example, we specify a line width of 12 characters. If no width is specified, the default is 80 characters. Notice how the lines are broken regardless of word boundaries. The addition of the option will cause to break the line at the last available space before the line width is reached:

```
                    J+  ,                    -!   J
0
)  A
  8
  6
    B,
  <
```

– A Simple Text Formatter

The program also folds text, plus a lot more. It accepts either files or standard input and performs paragraph formatting on the text stream. Basically, it fills and joins lines in text while preserving blank lines and indentation.

To demonstrate, we'll need some text. Let's lift some from the info page:

```
  I  J          6      @='       M          6
          O>    8              6  <

  G,      >        >  6      8   8   >
```

```
6            6  T          6       8
                T         6        6
    6  <

  I   J 6                              >
                         8
    8            <  3 _          _
      6     6     8            ,   I<EFJ>     8    ,  8
 6             >          ,          6
A    <  =    ) V> I   J           _6     6  _
        T                                , 9     '<
1       -       < :       _G       :     6   @    =    _>
I(   8     :       Y ' 6      J ""> "" MK       "%!"O> """% ""!&<
```

We'll copy this text into our text editor and save the file as < . Now, let's say we wanted to reformat this text to fit a fifty character wide column. We could do this by processing the file with and the 8 option:

```
                              "*          0
  I   J            6       @='
  M          6
         O>    8              6  <

  G,      >         >  6      8   8   >

6            6  T          6
8                           T
    6        6               6  <
```

Well, that's an awkward result. Perhaps we should actually read this text, since it explains what's going on:

```
  9       & !        &       !         &
      K                                          A     K   !
                                  '
```

So, is preserving the indentation of the first line. Fortunately, provides an option to correct this:

```
                              "*
  I   J            6       @='
M          6              O>    8
         6  <
```

```
  G,      >         > 6      8   8   >
              6              6  T
         6       8
          T         6        6
            6  <

  I   J 6
      >
       8
   8            <  3 _          _
                      6     6
   8            ,    I<EFJ>     8
 , 8   6              >          ,
         6           A    <  =    ) V>
I   J           _6     6  _
        T
         , 9     '< 1        -       <
:       _G       :     6   @    =    _>
I(   8    :       Y ' 6      J ""> ""
MK        "%!"O> """% ""!&<
```

Much better. By adding the option, we now have the desired result.

has some interesting options:

! " ( )

Option	Description
	Operate in mode. This preserves the indentation of the first two lines of a paragraph. Subsequent lines are aligned with the indentation of the second line.
6	Only format those lines beginning with the prefix . After formatting, the contents of are prefixed to each reformatted line. This option can be used to format text in source code comments. For example, any programming language or configuration file that uses a “#” character to delineate a comment could be formatted by specifying 6 JZ J so that only the comments will be formatted. See the example below.
	Split-only mode. In this mode, lines will only be split to fit the specified column width. Short lines will not be joined to fill lines. This mode is useful when formatting text such as code where joining is not desired.
	Perform uniform spacing. This will apply traditional “typewriter-

	style” formatting to the text. This means a single space between words and two spaces between sentences. This mode is useful for removing “justification,” that is, text that has been padded with spaces to force alignment on both the left and right margins.
8	Format text to fit within a column characters wide. The default is 75 characters. Note: actually formats lines slightly shorter than the specified width to allow for line balancing.

The 6 option is particularly interesting. With it, we can format selected portions of a file, provided that the lines to be formatted all begin with the same sequence of characters. Many programming languages use the pound sign (#) to indicate the beginning of a comment and thus can be formatted using this option. Let’s create a file that simulates a program that uses comments:

```
                    &
Z )          8      <

Z )         <
Z   8  ,          <
Z 3     <

)  >        >         <
3           <
3     <
```

Our sample file contains comments which begin with the string “# “ (a # followed by a space) and lines of “code” which do not. Now, using , we can format the comments and leave the code untouched:

```
                     "*   %U %
Z )          8      <

Z )         <   8  ,
Z       < 3     <

)  >        >         <
3           <
3     <
```

Notice that the adjoining comment lines are joined, while the blank lines and the lines that do not begin with the specified prefix are preserved.

6 – Format Text For Printing

The 6 program is used to text. When printing text, it is often desirable to separate the pages of output with several lines of whitespace, to provide a top and bottom margin for each page. Further, this whitespace can be used to insert a header and footer on each page.

We'll demonstrate 6 by formatting our < file into a series of very short pages (only the first two pages are shown):

```
                                    "   N"



*$"+ "* "" "! *0                    <              :   "



(2('          "$<*          "*5$05*$$+
              "$            ""5*#5*$$!
(2('          ""<$          $+5"%5*$$!
2             !<$&          $&5*&5*$$!
              !             ""5$!5*$$0



*$"+ "* "" "! *0                    <              :   *



(2('          "$</          "$5$&5*$$0
2             +<"$          "$5*+5*$$+
              0             $#5/"5*$$0
2             0<"$          "$5"!5*$$0
2             0<$&          $&5"%5*$$0
```

In this example, we employ the option (for page length) and the 8 option (page width) to define a "page" that is 65 columns wide and 15 lines long. 6 paginates the contents of the < file, separates each page with several lines of whitespace and creates a default header containing the file modification time, filename, and page number. The 6 program provides many options to control page layout. We'll take a look at more of them in the next chapter.

6 – Format And Print Data

Unlike the other commands in this chapter, the 6 command is not used for pipelines (it does not accept standard input) nor does it find frequent application directly on the command line (it's mostly used in scripts). So why is it important? Because it is so widely used.

6 (from the phrase "print formatted") was originally developed for the C programming language and has been implemented in many programming languages including the shell. In fact, in , 6 is a builtin.

6 works like this:

6 c d

The command is given a string containing a format description which is then applied to a list of arguments. The formatted result is sent to standard output. Here is a trivial example:

```
                    JP          2 8 K J
@
```

The format string may contain literal text (like "I formatted the string:"), escape sequences (such as ^ , a newline character), and sequences beginning with the 4 character, which are called . In the example above, the conversion specification 4 is used to format the string "foo" and place it in the command's output. Here it is again:

```
                    JP      %8 %        K J
@       J  J       <
```

As we can see, the 4 conversion specification is replaced by the string "foo" in the command's output. The s conversion is used to format string data. There are other specifiers for other kinds of data. This table lists the commonly used data types:

! " - . #

Specifier	Description
	Format a number as a signed decimal integer.
	Format and output a floating point number.
	Format an integer as an octal number.

	Format a string.
	Format an integer as a hexadecimal number using lowercase a-f where needed.
V	Same as but use uppercase letters.
4	Print a literal % symbol (i.e., specify “%%”)

We’ll demonstrate the effect each of the conversion specifiers on the string “380”:

```
                                J8 ; 8 ; 8 ; 8 ; 8ZK J  M*  M*  M*  M*
 M*  M*
/!$> /!$<$$$$$$> #0&> /!$> "0 > "0
```

Since we specified six conversion specifiers, we must also supply six arguments for 6 to process. The six results show the effect of each specifier.

Several optional components may be added to the conversion specifier to adjust its output. A complete conversion specification may consist of the following:

4 N 6

Multiple optional components, when used, must appear in the order specified above to be properly interpreted. Here is a description of each:

! " 2 #

Component	Description
	There are five different flags:
	Z – Use the “alternate format” for output. This varies by data type. For (octal number) conversion, the output is prefixed with $. For and V (hexadecimal number) conversions, the output is prefixed with $ or $V respectively.
	$W(zero) Pad the output with zeros. This means that the field will be filled with leading zeros, as in “000380”.
	– (dash) Left-align the output. By default, 6 right-aligns output.
	‘ ’ – (space) Produce a leading space for positive numbers.
	7 – (plus sign) Sign positive numbers. By default, 6 only

	signs negative numbers.
	A number specifying the minimum field width.
	For floating point numbers, specify the number of digits of precision to be output after the decimal point. For string conversion, specifies the number of characters to output.

Here are some examples of different formats in action:

! " 3 # 1

Argument	Format	Result	Notes
/!$	_4 _	/!$	Simple formatting of an integer.
/!$	_4Z _	$ "0	Integer formatted as a hexadecimal number using the “alternate format” flag.
/!$	_4$# _	$$/!$	Integer formatted with leading zeros (padding) and a minimum field width of five characters.
/!$	_4$#<# _	/!$<$$$$$	Number formatted as a floating point number with padding and five decimal places of precision. Since the specified minimum field width (5) is less than the actual width of the formatted number, the padding has no effect.
/!$	_4$"$<# _	$/!$<$$$$$	By increasing the minimum field width to 10 the padding is now visible.
/!$	_47 _	7/!$	The + flag signs a positive number.
/!$	_4 _	/!$	The - flag left aligns the formatting.

_4# _	A string formatted with a minimum field width.
_4<# _	By applying precision to a string, it is truncated.

Again, 6 is used mostly in scripts where it is employed to format tabular data, rather than on the command line directly. But we can still show how it can be used to solve various formatting problems. First, let's output some fields separated by tab characters:

```
                    J8 K 8 K 8 K J
  "          *      /
```

By inserting ^ (the escape sequence for a tab), we achieve the desired effect. Next, some numbers with neat formatting:

```
                    JT  2 8*"  8 "   H    2 85 " K J  *C
  A "N D"   "MD
=   $"$0"       /<"&* Q          7/*#!%
```

This shows the effect of minimum field width on the spacing of the fields. Or how about formatting a tiny web page:

```
                    J/   &K K /   &K K K /    &8 /    &K
K /    &K K /   !&K K K / &8 /  &K K /   !&K /    &K J JB   +
 J JB   =    J
X   C
     X   C
          X    C:   )   X5    C
     X5   C
     X   ,C
          X6C:        X56C
     X5   ,C
X5   C
```

Document Formatting Systems

So far, we have examined the simple text-formatting tools. These are good for small, sim-

ple tasks, but what about larger jobs? One of the reasons that Unix became a popular operating system among technical and scientific users (aside from providing a powerful multitasking, multiuser environment for all kinds of software development) is that it offered tools that could be used to produce many types of documents, particularly scientific and academic publications. In fact, as the GNU documentation describes, document preparation was instrumental to the development of Unix:

```
                0>+J                7.7 4                    9
 ! ' +   64                            7.7
          ' +        A                        &
                                    * L              '
                                    % +    M  N   M&        !  D'
 ' )     '
```

Two main families of document formatters dominate the field: those descended from the original program, including and , and those based on Donald Knuth's TEX (pronounced "tek") typesetting system. And yes, the dropped "E" in the middle is part of its name.

The name "roff" is derived from the term "run off" as in, "I'll run off a copy for you." The program is used to format documents for output to devices that use monospaced fonts, such as character terminals and typewriter-style printers. At the time of its introduction, this included nearly all printing devices attached to computers. The later program formats documents for output on , devices used to produce "camera-ready" type for commercial printing. Most computer printers today are able to simulate the output of typesetters. The family also includes some other programs that are used to prepare portions of documents. These include A (for mathematical equations) and (for tables).

The TEX system (in stable form) first appeared in 1989 and has, to some degree, displaced as the tool of choice for typesetter output. We won't be covering TEX here, due both to its complexity (there are entire books about it) and to the fact that it is not installed by default on most modern Linux systems.

(6 For those interested in installing TEX, check out the package which can be found in most distribution repositories, and the LyX graphical content editor.

is a suite of programs containing the GNU implementation of . It also includes a script that is used to emulate and the rest of the family as well.

While and its descendants are used to make formatted documents, they do it in a way that is rather foreign to modern users. Most documents today are produced using word processors that are able to perform both the composition and layout of a document in a single step. Prior to the advent of the graphical word processor, documents were often produced in a two-step process involving the use of a text editor to perform composition, and a processor, such as , to apply the formatting. Instructions for the formatting program were embedded into the composed text through the use of a markup language. The modern analog for such a process is the web page, which is composed using a text editor of some kind and then rendered by a web browser using HTML as the markup language to describe the final page layout.

We're not going to cover in its entirety, as many elements of its markup language deal with rather arcane details of typography. Instead we will concentrate on one of its that remains in wide use. These macro packages condense many of its low-level commands into a smaller set of high-level commands that make using much easier.

For a moment, let's consider the humble man page. It lives in the 5 5 5 directory as a B 6 compressed text file. If we were to examine its uncompressed contents, we would see the following (the man page for in section 1 is shown):

```
                                -                          - 0
<^_ 9S KS) -S9@ U )R@(  @='F  @  8            ,    6*    "</#<
<)R =( _"_ _36    *$$!_ _?K2          +<"$_ _2          _
<(R K3-'
  ^              ,
<(R (UKS:(@(
<G
 ^ @S:)@SK^ Q <<<  ^ @ @='^ Q <<<
<(R 9'( Q@:)@SK
<^_ 3     ,                6
<::
```

Compared to the man page in its normal presentation, we can begin to see a correlation between the markup language and its results:

```
                                  0
=(M"O                  2                          =(M"O


K3-'
                  ,
```

```
(UKS:(@(
        S:)@SK <<<   @=' <<<
```

The reason this is of interest is that man pages are rendered by , using the macro package. In fact, we can simulate the command with the following pipeline:

```
                        -                    - 0          +
    0
=(M"O                2                    =(M"O



K3-'
                ,

(UKS:(@(
        S:)@SK <<<   @=' <<<
```

Here we use the program with the options set to specify the macro package and the output driver for ASCII. can produce output in several formats. If no format is specified, PostScript is output by default:

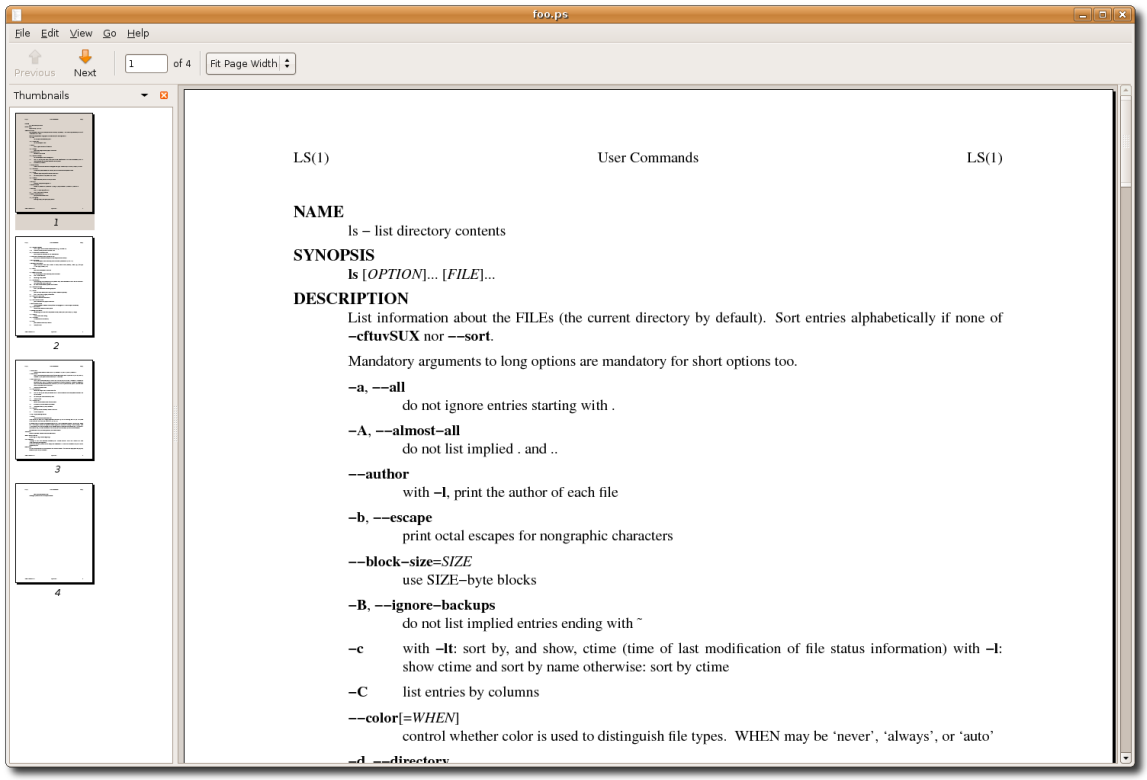
```
                        -                    - 0          0

4F:( 3    /<$
44                "<"!<"
44      9    )      # "/ && /0 *$$%
449     K    Q            )    Q
447     )    G
447     )    @
449     ( 66   Q        6        6  "<"! "
44:     &
44:  S     3
44S         :
```

We briefly mentioned PostScript in the previous chapter, and will again in the next chapter. PostScript is a page description language that is used to describe the contents of a printed page to a typesetter-like device. If we take the output of our command and store it to a file (assuming that we are using a graphical desktop with a 9 6 directory):

```
                    -                  - 0          &
1
```

An icon for the output file should appear on the desktop. By double-clicking the icon, a page viewer should start up and reveal the file in its rendered form:



- B 7 #) / * 7 B + ?>)%1

What we see is a nicely typeset man page for ! In fact, it's possible to convert the PostScript file into a PDF (7 ! .) file with this command:

```
                    1          1
```

The 6 *6 program is part of the 6 package, which is installed on most Linux systems that support printing.

(: Linux systems often include many command line programs for file format

conversion. They are often named using the convention of 2 . Try using the command 5 5 5D 6 * 6 D to identify them. Also try searching for programs named .

For our last exercise with , we will revisit our old friend < once more. This time, we will use the program which is used to format tables to typeset our list of Linux distributions. To do this, we are going to use our earlier script to add markup to a text stream that we will feed to .

First, we need to modify our script to add the necessary markup elements (called = in) that requires. Using a text editor, we will change < to the following:

```
Z      6    6     =              6

  K
 +FK
       #K
    K
      K
    K
T    1          H    K
$K
?            ]              H      K
.
 5^M $ % ^\*^]^O^5^M $ % ^\*^]^O^5^M $ % ^\&^]^O 5^/ ^" ^*5
3  K
 +G
```

Note that for the script to work properly, care must been taken to see that the words “Name Version Released” are separated by tabs, not spaces. We’ll save the resulting file as < . uses the <)(and <)' requests to start and end the table. The rows following the <)(request define global properties of the table which, for our example, are centered horizontally on the page and surrounded by a box. The remaining lines of the definition describe the layout of each table row. Now, if we run our report-generating pipeline again with the new script, we’ll get the following :

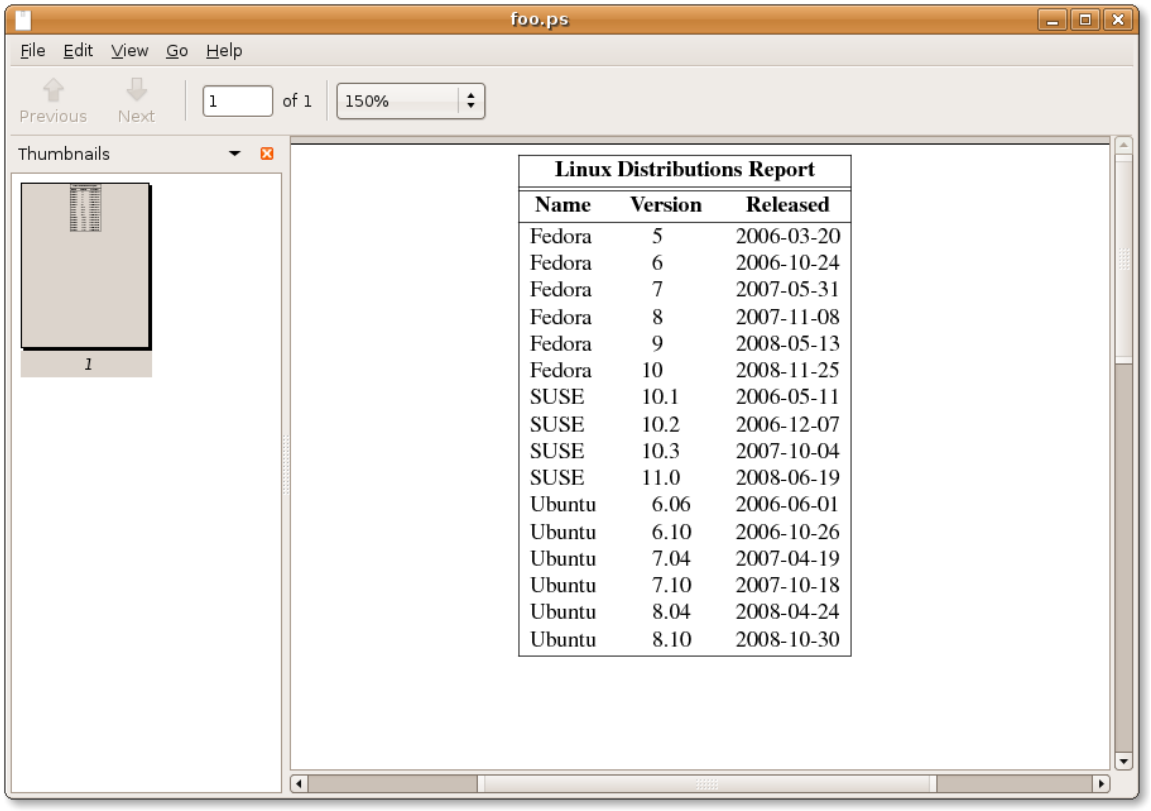
```
                          ;              0
   0       +      &
          7                  7
          P              T    1          H                    P
          7                  7
```

```
P                ?     ]       H                        P
7                      7
P        #      *$$+ $/ *$ P
P        +      *$$+ "$ *& P
P        0      *$$0 $# /" P
P        !      *$$0 "" $! P
P        %      *$$! $# "/ P
P        "$     *$$! "" *# P
P(2('    "$<"    *$$+ $# "" P
P(2('    "$<*    *$$+ "* $0 P
P(2('    "$</    *$$0 "$ $& P
P(2('    ""<$    *$$! $+ "% P
P2       +<$+    *$$+ $+ $" P
P2       +<"$    *$$+ "$ *+ P
P2       0<$&    *$$0 $& "% P
P2       0<"$    *$$0 "$ "! P
P2       !<$&    *$$! $& *& P
P2       !<"$    *$$! "$ /$ P
7                      7
```

Adding the option to instructs it to pre-process the text stream with . Likewise, the) option is used to output to ASCII rather than the default output medium, PostScript.

The format of the output is the best we can expect if we are limited to the capabilities of a terminal screen or typewriter-style printer. If we specify PostScript output and graphically view the output, we get a much more satisfying result:

```
                    ;          0
0     & 1
```



Linux Distributions Report		
Name	**Version**	**Released**
Fedora	5	2006-03-20
Fedora	6	2006-10-24
Fedora	7	2007-05-31
Fedora	8	2007-11-08
Fedora	9	2008-05-13
Fedora	10	2008-11-25
SUSE	10.1	2006-05-11
SUSE	10.2	2006-12-07
SUSE	10.3	2007-10-04
SUSE	11.0	2008-06-19
Ubuntu	6.06	2006-06-01
Ubuntu	6.10	2006-10-26
Ubuntu	7.04	2007-04-19
Ubuntu	7.10	2007-10-18
Ubuntu	8.04	2008-04-24
Ubuntu	8.10	2008-10-30

2 B !

Summing Up

Given that text is so central to the character of Unix-like operating systems, it makes sense that there would be many tools that are used to manipulate and format text. As we have seen, there are! The simple formatting tools like and 6 will find many uses in scripts that produce short documents, while (and friends) can be used to write books. We may never write a technical paper using command line tools (though there are many people who do!), but it's good to know that we could.

Further Reading

User's Guide
http://www.gnu.org/software/groff/manual/

/ 7 / 0 :
http://docs.freebsd.org/44doc/usd/19.memacros/paper.pdf

: % :

http://docs.freebsd.org/44doc/usd/20.meref/paper.pdf

! O * 7 ! :
http://plan9.bell-labs.com/10thEdMan/tbl.pdf

And, of course, try the following articles at Wikipedia:
http://en.wikipedia.org/wiki/TeX
http://en.wikipedia.org/wiki/Donald_Knuth
http://en.wikipedia.org/wiki/Typesetting

2

After spending the last couple of chapters manipulating text, it's time to put that text on paper. In this chapter, we'll look at the command line tools that are used to print files and control printer operation. We won't be looking at how to configure printing, as that varies from distribution to distribution and is usually set up automatically during installation. Note that we will need a working printer configuration to perform the exercises in this chapter.

We will discuss the following commands:

6 – Convert text files for printing

6 – Print files

*6 – Format files for printing on a PostScript printer

6 – Show printer status information

6A – Show printer queue status

6 – Cancel print jobs

A Brief History Of Printing

To fully understand the printing features found in Unix-like operating systems, we must first learn some history. Printing on Unix-like systems goes way back to the beginning of the operating system itself. In those days, printers and how they were used was much different from today.

Printing In The Dim Times

Like the computers themselves, printers in the pre-PC era tended to be large, expensive, and centralized. The typical computer user of 1980 worked at a terminal connected to a computer some distance away. The printer was located near the computer and was under the watchful eyes of the computer's operators.

When printers were expensive and centralized, as they often were in the early days of Unix, it was common practice for many users to share a printer. To identify print jobs be-

longing to a particular user, a ! displaying the name of the user was often printed at the beginning of each print job. The computer support staff would then load up a cart containing the day's print jobs and deliver them to the individual users.

Character-based Printers

The printer technology of the 80s was very different in two respects. First, printers of that period were almost always . Impact printers use a mechanical mechanism which strikes a ribbon against the paper to form character impressions on the page. Two of the popular technologies of that time were printing and printing.

The second, and more important characteristic of early printers was that printers used a fixed set of characters that were intrinsic to the device itself. For example, a daisy-wheel printer could only print the characters actually molded into the petals of the daisy wheel. This made the printers much like high-speed typewriters. As with most typewriters, they printed using monospaced (fixed width) fonts. This means that each character has the same width. Printing was done at fixed positions on the page, and the printable area of a page contained a fixed number of characters. Most printers printed ten characters per inch (CPI) horizontally and six lines per inch (LPI) vertically. Using this scheme, a US-letter sheet of paper is 85 characters wide and 66 lines high. Taking into account a small margin on each side, 80 characters was considered the maximum width of a print line. This explains why terminal displays (and our terminal emulators) are normally 80 characters wide. It provides a /C#+/C? (/ C # + / C ?) view of printed output, using a monospaced font.

Data is sent to a typewriter-like printer in a simple stream of bytes containing the characters to be printed. For example, to print an "a", the ASCII character code 97 is sent. In addition, the low-numbered ASCII control codes provided a means of moving the printer's carriage and paper, using codes for carriage return, line feed, form feed, etc. Using the control codes, it's possible to achieve some limited font effects, such as boldface, by having the printer print a character, backspace, and print the character again to get a darker print impression on the page. We can actually witness this if we use to render a man page and examine the output using 3 :

```
                    -                    - 0        0
 : 0
=(M"O              2                    =(M"O



K`RK3`R3-`R-'`R'
                ,
```

```
(`R(U`RUK`RKS`RS:`R:(`R(@`R@(`R(
      `R  `R   N`RSN`R:N`R)N`R@N`RSN`RK <<<  N`R N`R@N`R=N`R' <<<
```

The `R (Control-h) characters are the backspaces used to create the boldface effect. Likewise, we can also see a backspace/underscore sequence used to produce underlining.

Graphical Printers

The development of GUIs led to major changes in printer technology. As computers moved to more picture-based displays, printing moved from character-based to graphical techniques. This was facilitated by the advent of the low-cost laser printer which, instead of printing fixed characters, could print tiny dots anywhere in the printable area of the page. This made printing proportional fonts (like those used by typesetters), and even photographs and high-quality diagrams, possible.

However, moving from a character-based scheme to a graphical scheme presented a formidable technical challenge. Here's why: The number of bytes needed to fill a page using a character-based printer can be calculated this way (assuming 60 lines per page each containing 80 characters):

60 X 80 = 4800 bytes

In comparison, a 300 dot per inch (DPI) laser printer (assuming an 8 by 10 inch print area per page) requires:

(8 X 300) X (10 X 300) / 8 = 900000 bytes

Many of the slow PC networks simply could not handle the nearly one megabyte of data required to print a full page on a laser printer, so it was clear that a clever invention was needed.

That invention turned out to be the (PDL). A page description language is a programming language that describes the contents of a page. Basically it says, "go to this position, draw the character 'a' in 10 point Helvetica, go to this position..." until everything on the page is described. The first major PDL was 7 # from Adobe Systems, which is still in wide use today. The PostScript language is a complete programming language tailored for typography and other kinds of graphics and imaging. It includes built-in support for 35 standard, high-quality fonts, plus the ability to accept additional font definitions at run time. At first, support for PostScript was built into the printers themselves. This solved the data transmission problem. While the typical PostScript program was very verbose in comparison to the simple byte stream of character-based printers, it was much smaller than the number of bytes required to represent the entire printed page.

A 7 # accepted a PostScript program as input. The printer contained its

own processor and memory (oftentimes making the printer a more powerful computer than the computer to which it was attached) and executed a special program called a 7 # , which read the incoming PostScript program and the results into the printer's internal memory, thus forming the pattern of bits (dots) that would be transferred to the paper. The generic name for this process of rendering something into a large bit pattern (called a !) is or RIP.

As the years went by, both computers and networks became much faster. This allowed the RIP to move from the printer to the host computer, which, in turn, permitted high-quality printers to be much less expensive.

Many printers today still accept character-based streams, but many low-cost printers do not. They rely on the host computer's RIP to provide a stream of bits to print as dots. There are still some PostScript printers, too.

Printing With Linux

Modern Linux systems employ two software suites to perform and manage printing. The first, CUPS (Common Unix Printing System) provides print drivers and print-job management, and the second, Ghostscript, a PostScript interpreter, acts as a RIP.

CUPS manages printers by creating and maintaining = . As we discussed in our history lesson above, Unix printing was originally designed to manage a centralized printer shared by multiple users. Since printers are slow by nature, compared to the computers that are feeding them, printing systems need a way to schedule multiple print jobs and keep things organized. CUPS also has the ability to recognize different types of data (within reason) and can convert files to a printable form.

Preparing Files For Printing

As command line users, we are mostly interested in printing text, though it is certainly possible to print other data formats as well.

6 – Convert Text Files For Printing

We looked at 6 a little in the previous chapter. Now we will examine some of its many options used in conjunction with printing. In our history of printing, we saw how character-based printers use monospaced fonts, resulting in fixed numbers of characters per line and lines per page. 6 is used to adjust text to fit on a specific page size, with optional page headers and margins. Here is a summary of its most commonly used options:

! "")

Option	Description

7	Output a range of pages starting with and, optionally, ending with .
	Organize the content of the page into the number of columns specified by .
	By default, multicolumn output is listed vertically. By adding the -a (across) option, content is listed horizontally.
	Double-space output.
9 c d	Format the date displayed in page headers using . See the man page for the command for a description of the format string.
	Use form feeds rather than carriage returns to separate pages.
c d	In the center portion of the page header, use rather than the name of the file being processed.
	Set page length to . Default is 66 (US letter at 6 lines per inch)
	Number lines.
	Create a left margin characters wide.
8	Set page width to . Default is 72.

6 is often used in pipelines as a filter. In this example, we will produce a directory listing of 5 5 and format it into paginated, three-column output using 6 :

```
                                0        N" 0


*$"+ $* "! "& $$                          :   "



                        6                          8
&""  66
 *6
 *6                                                  ,
 *6   6  8  66
```

Sending A Print Job To A Printer

The CUPS printing suite supports two methods of printing historically used on Unix-like systems. One method, called Berkeley or LPD (used in the Berkeley Software Distribution version of Unix), uses the 6 program, while the other method, called SysV (from the System V version of Unix), uses the 6 program. Both programs do roughly the same thing. Choosing one over the other is a matter of personal taste.

6 – Print Files (Berkeley Style)

The 6 program can be used to send files to the printer. It may also be used in pipelines, as it accepts standard input. For example, to print the results of our multicolumn directory listing above, we could do this:

```
                    0  0
```

and the report would be sent to the system's default printer. To send the file to a different printer, the : option can be used like this:

```
 B
```

where $ is the name of the desired printer. To see a list of printers known to the system:

```

```

(6 Many Linux distributions allow you to define a "printer" that outputs files in PDF (Portable Document Format), rather than printing on the physical printer. This is very handy for experimenting with printing commands. Check your printer configuration program to see if it supports this configuration. On some distributions, you may need to install additional packages (such as 6 6) to enable this capability.

Here are some of the common options for 6 :

! "" ")

Option	Description
Z	Set number of copies to ! .
6	Print each page with a shaded header with the date, time, job name, and page number. This so-called “pretty print” option can be used when printing text files.
:	Specify the name of the printer used for output. If no printer is specified, the system’s default printer is used.
	Delete files after printing. This would be useful for programs that produce temporary printer-output files.

6 – Print Files (System V Style)

Like 6 , 6 accepts either files or standard input for printing. It differs from 6 in that it supports a different (and slightly more sophisticated) option set. Here are the common options:

! "" ( )

Option	Description
	Set the destination (printer) to . If no d option is specified, the system default printer is used.
	Set the number of copies to ! .
6	Set output to landscape orientation.
6	Scale the file to fit the page. This is useful when printing images, such as JPEG files.
L	Scale file to ! . The value of 100 fills the page. Values less than 100 are reduced, while values greater than 100 cause the file to be printed across multiple pages.
6 L	Set the output characters per inch to ! . Default is 10.
6 L	Set the output lines per inch to ! . Default is 6.

6 L 6 L 6 L 6 6L	Set the page margins. Values are expressed in , a unit of typographic measurement. There are 72 points to an inch.
:	Specify the list of pages. may be expressed as a comma-separated list and/or a range. For example “1,3,5,7-10”

We’ll produce our directory listing again, this time printing 12 CPI and 8 LPI with a left margin of one half inch. Note that we have to adjust the 6 options to account for the new page size:

```
                    0   A   D*   MM 0          $ N
  $      $M
```

This pipeline produces a four-column listing using smaller type than the default. The increased number of characters per inch allows us to fit more columns on the page.

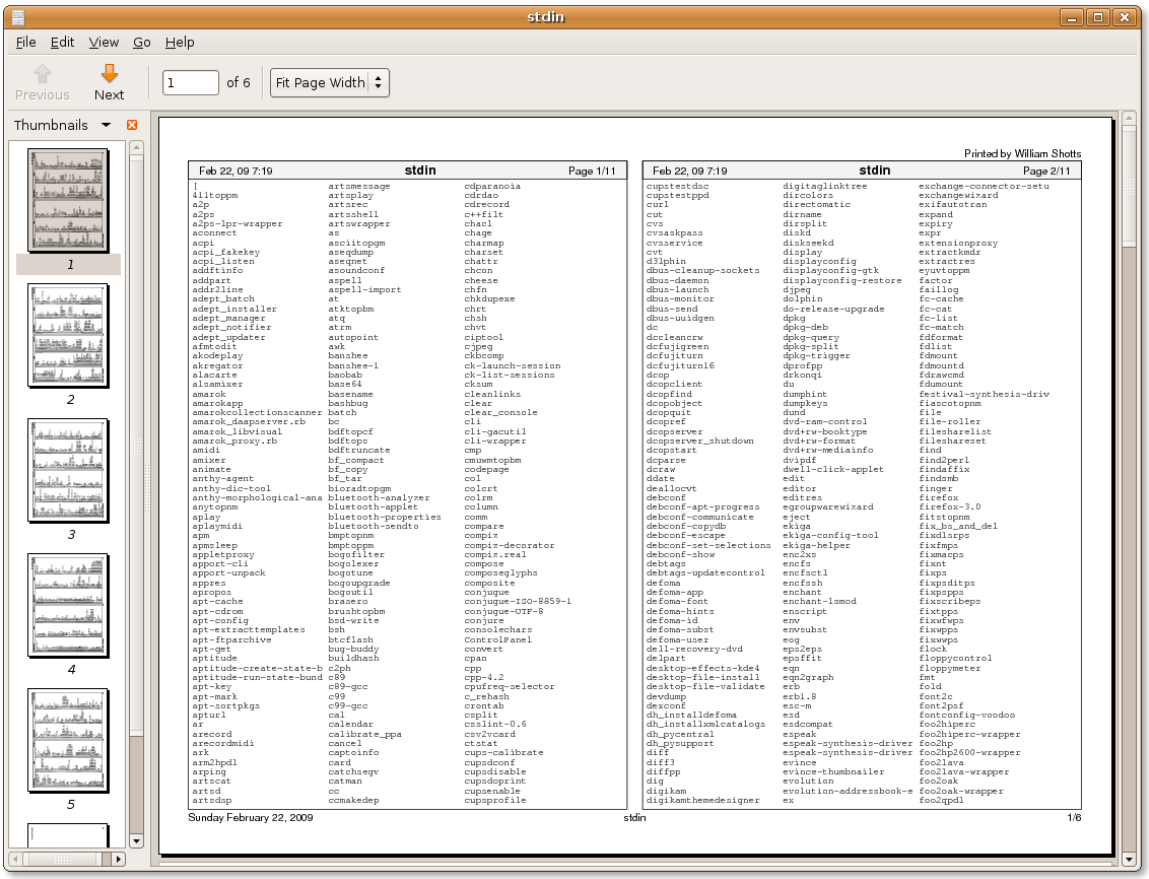
Another Option: *6

The *6 program is interesting. As we can surmise from its name, it’s a format conversion program, but it also much more. Its name originally meant “ASCII to PostScript” and it was used to prepare text files for printing on PostScript printers. Over the years, however, the capabilities of the program have grown, and now its name means “Anything to PostScript.” While its name suggests a format-conversion program, it is actually a printing program. It sends its default output to the system’s default printer rather than standard output. The program’s default behavior is that of a “pretty printer,” meaning that it improves the appearance of output. If we use the program to create a PostScript file on our desktop:

```
                    0      0      1        T
NN
    M6   O  "" 6      +
 )    "" 6      +              I5   5  59    65
 <6 J
```

Here we filter the stream with 6 , using the option (omit headers and footers) and then with *6 , specifying an output file (option) and 66 lines per page (= option)

to match the output pagination of 6 . If we view the resulting file with a suitable file viewer, we will see this:



3 B ")

As we can see, the default output layout is “two up” format. This causes the contents of 2 pages to be printed on each sheet of paper. *6 applies nice page headers and footers, too.

*6 has a lot of options. Here is a summary:

! "" - ")

Option	Description
L	Set center page title to .
L	Arrange pages into ! columns. Default is 2.

L	Set page footer to .
	Report the types of files given as arguments. Since *6 tries to convert and format all types of data, this option can be useful for predicting what *6 will do when given a particular file.
L	Set left-page footer to .
L	Set left-page title to .
L	Number lines of output every lines.
L	Display default settings.
L	Display settings for , where is one of the following: delegations (external programs that will be used to convert data), encodings, features, variables, media (paper sizes and the like), ppd (PostScript printer descriptions), printers, prologues (portions of code that are prefixed to normal output), stylesheets, and user options.
6 L	Print pages in range.
L	Set right-page footer to .
L	Set right-page title to .
8 L	Arrange pages into ! rows. Default is one.
G	No page headers.
	Set page header to .
	Use 8 point font.
	Set characters per line to ! . This and the = option (below) can be used to make files paginated with other programs, such as 6 , fit correctly on the page.
=	Set lines per page to ! .
-	Use media name. For example, “A4”.
	Output ! copies of each page.

	Send output to . If is specified as “-”, use standard output.
:	Use . If a printer is not specified, the system default printer is used.
Q	Portrait orientation.
	Landscape orientation.
)	Set tab stops to every ! characters.
	Underlay (watermark) pages with .

This is just a summary. *6 has several more options.

2 6 There is another output formatter that is useful for converting text into PostScript. Called 6 , it can perform many of the same kinds of formatting and printing tricks, but unlike *6 , it only accepts text input.

Monitoring And Controlling Print Jobs

As Unix printing systems are designed to handle multiple print jobs from multiple users, CUPS is designed to do the same. Each printer is given a = , where jobs are parked until they can be to the printer. CUPS supplies several command line programs that are used to manage printer status and print queues. Like the 6 and 6 programs, these management programs are modeled after the corresponding programs from the Berkeley and System V printing systems.

6 – Display Print System Status

The 6 program is useful for determining the names and availability of printers on the system. For example, if we had a system with both a physical printer (named “printer”) and a PDF virtual printer (named “PDF”), we could check their status like this:

```
:9   6    A        -  $! 9  *$"# $/ $# #% :- '()
6      6    A        )  *&   *$"+ $! &/ ** 3- '()
```

Further, we could determine a more detailed description of the print system configuration this way:

```
,                6
        :9    6  6   5
        6       66 556          +/"56      56
```

In this example, we see that “printer” is the system’s default printer and that it is a network printer using Internet Printing Protocol (ipp://) attached to a system named “printserver”.

The commonly useful options include:

! "" 2)

Option	Description
<<<	Display the state of the printer queue for . Note that this is the status of the printer queue’s ability to accept jobs, not the status of the physical printers. If no printers are specified, all print queues are shown.
	Display the name of the system’s default printer.
6 <<<	Display the status of the specified . If no printers are specified, all printers are shown.
	Display the status of the print server.
	Display a status summary.
	Display a complete status report.

6A – Display Printer Queue Status

To see the status of a printer queue, the 6A program is used. This allows us to view the status of the queue and the print jobs it contains. Here is an example of an empty queue for a system default printer named “printer”:

```
                    ,
6          ,
```

If we do not specify a printer (using the : option), the system’s default printer is shown. If we send a job to the printer and then look at the queue, we will see it listed:

```
                                0      0
  A         6      +$/ M"     M OO
                              ,
6           ,    6
Q      S8              M O                  )    ( B
            +$/    M     O                  "$*&  ,
```

6 / – Cancel Print Jobs

CUPS supplies two programs used to terminate print jobs and remove them from the print queue. One is Berkeley style (6) and the other is System V ( ). They differ slightly in the options they support, but do basically the same thing. Using our print job above as an example, we could stop the job and remove it this way:

```
                              N*
                              ,
6           ,
```

Each command has options for removing all the jobs belonging to a particular user, particular printer, and multiple job numbers. Their respective man pages have all the details.

Summing Up

In this chapter, we have seen how the printers of the past influenced the design of the printing systems on Unix-like machines, and how much control is available on the command line to control not only the scheduling and execution of print jobs, but also the various output options.

Further Reading

A good article on the PostScript page description language:
http://en.wikipedia.org/wiki/PostScript

The Common Unix Printing System (CUPS):
http://en.wikipedia.org/wiki/Common_Unix_Printing_System
http://www.cups.org/

The Berkeley and System V Printing Systems:
http://en.wikipedia.org/wiki/Berkeley_printing_system
http://en.wikipedia.org/wiki/System_V_printing_system

$" 2 $

In this chapter, we will look at how to build programs by compiling source code. The availability of source code is the essential freedom that makes Linux possible. The entire ecosystem of Linux development relies on free exchange between developers. For many desktop users, compiling is a lost art. It used to be quite common, but today, distribution providers maintain huge repositories of precompiled binaries, ready to download and use. At the time of this writing, the Debian repository (one of the largest of any of the distributions) contains almost 23,000 packages.

So why compile software? There are two reasons:

1. # . Despite the number of precompiled programs in distribution repositories, some distributions may not include all the desired applications. In this case, the only way to get the desired program is to compile it from source.
2. . While some distributions specialize in cutting edge versions of programs, many do not. This means that in order to have the very latest version of a program, compiling is necessary.

Compiling software from source code can become very complex and technical; well beyond the reach of many users. However, many compiling tasks are quite easy and involve only a few steps. It all depends on the package. We will look at a very simple case in order to provide an overview of the process and as a starting point for those who wish to undertake further study.

We will introduce one new command:

– Utility to maintain programs

What Is Compiling?

Simply put, compiling is the process of translating (the human-readable description of a program written by a programmer) into the native language of the computer's processor.

The computer's processor (or 70) works at a very elemental level, executing programs in what is called . This is a numeric code that describes very small operations, such as "add this byte," "point to this location in memory," or "copy this byte."

Each of these instructions is expressed in binary (ones and zeros). The earliest computer programs were written using this numeric code, which may explain why programmers who wrote it were said to smoke a lot, drink gallons of coffee, and wear thick glasses.

This problem was overcome by the advent of ! , which replaced the numeric codes with (slightly) easier to use character such as CPY (for copy) and MOV (for move). Programs written in assembly language are processed into machine language by a program called an ! . Assembly language is still used today for certain specialized programming tasks, such as and ! .

We next come to what are called . They are called this because they allow the programmer to be less concerned with the details of what the processor is doing and more with solving the problem at hand. The early ones (developed during the 1950s) included): :*> (designed for scientific and technical tasks) and)9) (designed for business applications). Both are still in limited use today.

While there are many popular programming languages, two predominate. Most programs written for modern systems are written in either or PP . In the examples to follow, we will be compiling a C program.

Programs written in high-level programming languages are converted into machine language by processing them with another program, called a . Some compilers translate high-level instructions into assembly language and then use an assembler to perform the final stage of translation into machine language.

A process often used in conjunction with compiling is called . There are many common tasks performed by programs. Take, for instance, opening a file. Many programs perform this task, but it would be wasteful to have each program implement its own routine to open files. It makes more sense to have a single piece of programming that knows how to open files and to allow all programs that need it to share it. Providing support for common tasks is accomplished by what are called ! . They contain multiple , each performing some common task that multiple programs can share. If we look in the 5 and 5 5 directories, we can see where many of them live. A program called a is used to form the connections between the output of the compiler and the libraries that the compiled program requires. The final result of this process is the ! , ready for use.

Are All Programs Compiled?

No. As we have seen, there are programs such as shell scripts that do not require compiling. They are executed directly. These are written in what are known as or languages. These languages have grown in popularity in recent years and include 7 &7 & 7,7& : ! , and many others.

Scripted languages are executed by a special program called an . An interpreter inputs the program file and reads and executes each instruction contained within it. In

general, interpreted programs execute much more slowly than compiled programs. This is because each source code instruction in an interpreted program is translated every time it is carried out, whereas with a compiled program, a source code instruction is only translated once, and this translation is permanently recorded in the final executable file.

So why are interpreted languages so popular? For many programming chores, the results are “fast enough,” but the real advantage is that it is generally faster and easier to develop interpreted programs than compiled programs. Programs are usually developed in a repeating cycle of code, compile, test. As a program grows in size, the compilation phase of the cycle can become quite long. Interpreted languages remove the compilation step and thus speed up program development.

Compiling A C Program

Let’s compile something. Before we do that however, we’re going to need some tools like the compiler, the linker, and . The C compiler used almost universally in the Linux environment is called (GNU C Compiler), originally written by Richard Stallman. Most distributions do not install by default. We can check to see if the compiler is present like this:

```
5 5 5
```

The results in this example indicate that the compiler is installed.

(: Your distribution may have a meta-package (a collection of packages) for software development. If so, consider installing it if you intend to compile programs on your system. If your system does not provide a meta-package, try installing the and packages. On many distributions, this is sufficient to carry out the exercise below.

Obtaining The Source Code

For our compiling exercise, we are going to compile a program from the GNU Project called . This is a handy little program that checks text files for writing quality and style. As programs go, it is fairly small and easy to build.

Following convention, we’re first going to create a directory for our source code named and then download the source code into it using 6 :

```
                                                  _____________
                                  6<   <          <
**$ ?K2  ):                ,<
K    M  6<   <      O                        !
*/$ =                 <
Q       ,      ,6     2K@V<
2          ,                      <
  6C
*#$ 9       ,            ,        <
  6C
*$$ :SQ)                    <                :3(;<
"#$ R                        ,        <
  8          " "$$/  +##/&   +!%&$ 3   *!  "%%!        $<0<   < B
  8          " "$$/  +##/&   %$%#0 -   $&  *$$*        "<$*<   < B
  8          " "$$/  +##/&  "&"$+* ( 6 "0  *$$0        "<""<   < B
**+ 9       ,      S1<
  6C                          -
             "<""<   < B                "<""<   < B
*$$ :SQ)                    <                :3(;<
"#$ S6      G@K3QU                                  "<""<   < B
M"&"$+*  ,   O<
**+           S1<
"&"$+*  ,                $<"+      M!&0<&  G5 O
  6C     !
**" ?    , <

       "<""<   < B
```

2 6 Since we are the “maintainer” of this source code while we compile it, we will keep it in 5 . Source code installed by your distribution will be installed in 5 5 , while source code we maintain that's intended for use by multiple users is usually installed in 5 5 5 .

As we can see, source code is usually supplied in the form of a compressed tar file. Sometimes called a ! , this file contains the , or hierarchy of directories and files that comprise the source code. After arriving at the ftp site, we examine the list of tar files available and select the newest version for download. Using the command within 6 , we copy the file from the ftp server to the local machine.

Once the tar file is downloaded, it must be unpacked. This is done with the program:

```
                                  -                -
```

```
"<""          "<""<   < B
```

(6 The program, like all GNU Project software, follows certain standards for source code packaging. Most other source code available in the Linux ecosystem also follows this standard. One element of the standard is that when the source code tar file is unpacked, a directory will be created which contains the source tree, and that this directory will be named A ' , thus containing both the project's name and its version number. This scheme allows easy installation of multiple versions of the same program. However, it is often a good idea to examine the layout of the tree before unpacking it. Some projects will not create the directory, but instead will deliver the files directly into the current directory. This will make a mess in your otherwise well-organized directory. To avoid this, use the following command to examine the contents of the tar file:

B P

Examining The Source Tree

Unpacking the tar file results in the creation of a new directory, named "<"" . This directory contains the source tree. Let's look inside:

```
                    "<""
   <          <          6 <
   < <         <6         6 <       <6
   <          < 6        6 N   <   Q'39-'
             < 6  <   @K()3==            <
     <        <   <                    <
S:U@K?                  -     <      ,  <"<
          N?G            <         ,  <
 <6        N?G<6           <
    <"<      6 "<       K'.(
```

In it, we see a number of files. Programs belonging to the GNU Project, as well as many others, will supply the documentation files Q'39-' , @K()3==, K'.(, and S:U@K?. These files contain the description of the program, information on how to build and install it, and its licensing terms. It is always a good idea to carefully read the Q'39-' and @K()3== files before attempting to build the program.

The other interesting files in this directory are the ones ending with < and < :

```
                    "<""
   <     6 "<     6 <     <        <    , <
                    "<""
  6 <     6 N  <     <        <
```

The < files contain the two C programs supplied by the package (, and), divided into modules. It is common practice for large programs to be broken into smaller, easier to manage pieces. The source code files are ordinary text and can be examined with :

```
                    "<""
```

The < files are known as . These, too, are ordinary text. Header files contain descriptions of the routines included in a source code file or library. In order for the compiler to connect the modules, it must receive a description of all the modules needed to complete the entire program. Near the beginning of the < file, we see this line:

```
Z     _   6 < _
```

This instructs the compiler to read the file 6 < as it reads the source code in < in order to “know” what’s in 6 < . The 6 < file supplies routines that are shared by both the , and programs.

Above the statement for 6 < , we see some other statements such as these:

```
Z     X    < C
Z     X    < C
Z     X     < C
Z     X     < C
Z     X     < C
```

These also refer to header files, but they refer to header files that live outside the current source tree. They are supplied by the system to support the compilation of every program. If we look in 5 5 , we can see them:

```
                    "<""
```

The header files in this directory were installed when we installed the compiler.

Building The Program

Most programs build with a simple, two-command sequence:

```
<5
```

The program is a shell script which is supplied with the source tree. Its job is to analyze the ! . Most source code is designed to be ! . That is, it is designed to build on more than one kind of Unix-like system. But in order to do that, the source code may need to undergo slight adjustments during the build to accommodate differences between systems. also checks to see that necessary external tools and components are installed. Let's run . Since is not located where the shell normally expects programs to be located, we must explicitly tell the shell its location by prefixing the command with <5 to indicate that the program is located in the current working directory:

```
                    "<""
```

will output a lot of messages as it tests and configures the build. When it finishes, it will look something like this:

```
          <  6     <<< ,
             < <<< ,
            ,            <<<      A
             <5    <
     <           -
     <              <"
     <              <
     <              < 6
     <           , <"
     <            5
     <              <
                    "<""
```

What's important here is that there are no error messages. If there were, the configuration failed, and the program will not build until the errors are corrected.

We see created several new files in our source directory. The most important one is - . - is a configuration file that instructs the program exactly how to build the program. Without it, will refuse to run. - is an ordinary text file, so we can view it:

```
                    "<""                O
```

The program takes as input a (which is normally named -), that describes the relationships and dependencies among the components that comprise the finished program.

The first part of the makefile defines variables that are substituted in later sections of the makefile. For example we see the line:

```
L
```

which defines the C compiler to be . Later in the makefile, we see one instance where it gets used:

```
        <       <     <     6 <     6 "<
    M  O      M=9 =3?(O      <       <     <  ^
      6 <     6 "<   M=@G(O
```

A substitution is performed here, and the value M O is replaced by at run time.

Most of the makefile consists of lines, which define a , in this case the executable file , and the files on which it is dependent. The remaining lines describe the command(s) needed to create the target from its components. We see in this example that the executable file (one of the final end products) depends on the existence of < , < , < , 6 < , and 6 "< . Later on, in the makefile, we see definitions of each of these as targets:

```
  <          <      <     6 <     <       <
6 <         6 <     6 <     6 N   <
6 "<        6 "<     6 <     6 N   <
<           <      <     <
```

```
     <          <      <     <       <
  , <         , <      <     6 <     <       <
```

However, we don't see any command specified for them. This is handled by a general target, earlier in the file, that describes the command used to compile any < file into a < file:

```
< <
            M  O    M :: =3?(O  M  =3?(O  X
```

This all seems very complicated. Why not simply list all the steps to compile the parts and be done with it? The answer to this will become clear in a moment. In the meantime, let's run and build our programs:

```
                        "<""
```

The program will run, using the contents of - to guide its actions. It will produce a lot of messages.

When it finishes, we will see that all the targets are now present in our directory:

```
                        "<""
    <        <6                                  <
    <                   N?G        -              <
    < <          <"        N?G<       -      <         <
    <            <"<       N?G<6          <        ,
    <            <          6 "<         <        ,  <"
    <            <          6 "<         <        ,  <"<
                <6          6 <      K'.(        ,  <
      <          < 6         6 <                 ,  <
 S:U@K?            < 6  <       6 N   <     <
                <           6 <        <6
  <              <    <    @K()3==      Q'39-'
```

Among the files, we see and , , the programs that we set out to build. Congratulations are in order! We just compiled our first programs from source code!

But just out of curiosity, let's run again:

```
                    "<""
   K              I  J<
```

It only produces this strange message. What's going on? Why didn't it build the program again? Ah, this is the magic of . Rather than simply building everything again, only builds what needs building. With all of the targets present, determined that there was nothing to do. We can demonstrate this by deleting one of the targets and running make again to see what it does. Let's get rid of one of the intermediate targets:

```
                    "<""
                    "<""
```

We see that rebuilds it and re-links the and , programs, since they depend on the missing module. This behavior also points out another important feature of : it keeps targets up to date. insists that targets be newer than their dependencies. This makes perfect sense, as a programmer will often update a bit of source code and then use to build a new version of the finished product. ensures that everything that needs building based on the updated code is built. If we use the program to "update" one of the source code files, we can see this happen:

```
                    "<""
 8      "          /0"+& *$$% $/ $# $+ "&
 8      "          //"*# *$$0 $/ /$ "0 &#     6 <
                    "<""
                    "<""
 8      "          /0"+& *$$% $/ $# $+ "&
 8      "          //"*# *$$% $/ $# $+ */     6 <
                    "<""
```

After runs, we see that it has restored the target to being newer than the dependency:

```
                    "<""
 8      "          /0"+& *$$% $/ $# $+ *&
 8      "          //"*# *$$% $/ $# $+ */     6 <
```

The ability of to intelligently build only what needs building is a great benefit to programmers. While the time savings may not be very apparent with our small project, it

is very significant with larger projects. Remember, the Linux kernel (a program that undergoes continuous modification and improvement) contains several lines of code.

Installing The Program

Well-packaged source code will often include a special target called . This target will install the final product in a system directory for use. Usually, this directory is 5 5 5 , the traditional location for locally built software. However, this directory is not normally writable by ordinary users, so we must become the superuser to perform the installation:

```
"<""
```

After we perform the installation, we can check that the program is ready to go:

```
"<""
5 5  5 5
"<""
```

And there we have it!

Summing Up

In this chapter, we have seen how three simple commands:

<5

can be used to build many source code packages. We have also seen the important role that plays in the maintenance of programs. The program can be used for any task that needs to maintain a target/dependency relationship, not just for compiling source code.

Further Reading

The Wikipedia has good articles on compilers and the program:
http://en.wikipedia.org/wiki/Compiler
http://en.wikipedia.org/wiki/Make_(software)

?>0 % % :
http://www.gnu.org/software/make/manual/html_node/index.html

Part 4 – Writing Shell Scripts

% 8 ' "

In the preceding chapters, we have assembled an arsenal of command line tools. While these tools can solve many kinds of computing problems, we are still limited to manually using them one by one on the command line. Wouldn't it be great if we could get the shell to do more of the work? We can. By joining our tools together into programs of our own design, the shell can carry out complex sequences of tasks all by itself. We can enable it to do this by writing .

What Are Shell Scripts?

In the simplest terms, a shell script is a file containing a series of commands. The shell reads this file and carries out the commands as though they have been entered directly on the command line.

The shell is somewhat unique, in that it is both a powerful command line interface to the system and a scripting language interpreter. As we will see, most of the things that can be done on the command line can be done in scripts, and most of the things that can be done in scripts can be done on the command line.

We have covered many shell features, but we have focused on those features most often used directly on the command line. The shell also provides a set of features usually (but not always) used when writing programs.

How To Write A Shell Script

To successfully create and run a shell script, we need to do three things:

1. ' (Shell scripts are ordinary text files. So we need a text editor to write them. The best text editors will provide , allowing us to see a color-coded view of the elements of the script. Syntax highlighting will help us spot certain kinds of common errors. , , , and many other editors are good candidates for writing scripts.
2. : (The system is rather fussy about not letting any old text file be treated as a program, and for good reason! We need to set the script file's permissions to allow execution.

3. (+ " The shell automatically searches certain directories for executable files when no explicit pathname is specified. For maximum convenience, we will place our scripts in these directories.

Script File Format

In keeping with programming tradition, we'll create a "hello world" program to demonstrate an extremely simple script. So let's fire up our text editors and enter the following script:

```
UL

U +

   %R    W   L%
```

The last line of our script is pretty familiar, just an command with a string argument. The second line is also familiar. It looks like a comment that we have seen used in many of the configuration files we have examined and edited. One thing about comments in shell scripts is that they may also appear at the ends of lines, like so:

```
   JR   .   FJ Z )
```

Everything from the Z symbol onward on the line is ignored.

Like many things, this works on the command line, too:

```
                        %R    W   L% U +
R   .   F
```

Though comments are of little use on the command line, they will work.

The first line of our script is a little mysterious. It looks as if it should be a comment, since it starts with Z, but it looks too purposeful to be just that. The ZF character sequence is, in fact, a special construct called a ! . The shebang is used to tell the system the name of the interpreter that should be used to execute the script that follows. Every shell script should include this as its first line.

Let's save our script file as N8 .

Executable Permissions

The next thing we have to do is make our script executable. This is easily done using :

```
                                .
8      "          +/ *$$% $/ $0 "$ "$     N8
                          C""     .
                                .
8      "          +/ *$$% $/ $0 "$ "$     N8
```

There are two common permission settings for scripts; 755 for scripts that everyone can execute, and 700 for scripts that only the owner can execute. Note that scripts must be readable in order to be executed.

Script File Location

With the permissions set, we can now execute our script:

```
                          .
R    .   F
```

In order for the script to run, we must precede the script name with an explicit path. If we don't, we get this:

```
                          .
       N8
```

Why is this? What makes our script different from other programs? As it turns out, nothing. Our script is fine. Its location is the problem. Back in Chapter 11, we discussed the :3)R environment variable and its effect on how the system searches for executable programs. To recap, the system searches a list of directories each time it needs to find an executable program, if no explicit path is specified. This is how the system knows to execute 5 5 when we type at the command line. The 5 directory is one of the directories that the system automatically searches. The list of directories is held within an environment variable named :3)R . The :3)R variable contains a colon-separated list of directories to be searched. We can view the contents of :3)R :

```
                        3B:+R
5   5 5   5  5    5    5  5    5   5  5    5  5   5
5   5  5
```

Here we see our list of directories. If our script were located in any of the directories in the list, our problem would be solved. Notice the first directory in the list, 5 5 5 . Most Linux distributions configure the :3)R variable to contain a directory in the user's home directory, to allow users to execute their own programs. So if we create the directory and place our script within it, it should start to work like other programs:

```
                          .
                         .
R    .   F
```

And so it does.

If the :3)R variable does not contain the directory, we can easily add it by including this line in our < file:

```
     B:+R$    2J3B:+RJ
```

After this change is made, it will take effect in each new terminal session. To apply the change to the current terminal session, we must have the shell re-read the < file. This can be done by "sourcing" it:

```

```

The dot (<) command is a synonym for the command, a shell builtin which reads a specified file of shell commands and treats it like input from the keyboard.

---

2 6 Ubuntu (and most other Debian-based distributions) automatically adds the 5 directory to the :3)R variable if the 5 directory exists when the user's < file is executed. So, on Ubuntu systems, if we create the 5 directory and then log out and log in again, everything works.

Good Locations For Scripts

The 5 directory is a good place to put scripts intended for personal use. If we write a script that everyone on a system is allowed to use, the traditional location is 5 5 5 . Scripts intended for use by the system administrator are often located in 5 5 5 . In most cases, locally supplied software, whether scripts or compiled programs, should be placed in the 5 5 hierarchy and not in 5 or 5 5 < These directories are specified by the Linux Filesystem Hierarchy Standard to contain only files supplied and maintained by the Linux distributor.

More Formatting Tricks

One of the key goals of serious script writing is ease of ; that is, the ease with which a script may be modified by its author or others to adapt it to changing needs. Making a script easy to read and understand is one way to facilitate easy maintenance.

Long Option Names

Many of the commands we have studied feature both short and long option names. For instance, the command has many options that can be expressed in either short or long form. For example:

```

```

and:

```
!
```

are equivalent commands. In the interests of reduced typing, short options are preferred when entering options on the command line, but when writing scripts, long options can provide improved readability.

Indentation And line-continuation

When employing long commands, readability can be enhanced by spreading the command over several lines. In Chapter 17, we looked at a particularly long example of the find command:

```
                          !     K4  !          *N**
    *N** `9>a `#a K6    K4  !          *C**
*C** `9>a `#a K6
```

Obviously, this command is a little hard to figure out at first glance. In a script, this command might be easier to understand if written this way:

```
6 ,     ^
  ^M ^
          ,6   ^
             6   $+$$ ^
                 $+$$ a\]b aTb ^
  ^O ^
     ^
  ^M ^
          ,6   ^
             6   $0$$ ^
                 $0$$ a\]b aTb ^
  ^O
```

By using line-continuations (backslash-linefeed sequences) and indentation, the logic of this complex command is more clearly described to the reader. This technique works on the command line, too, though it is seldom used, as it is very awkward to type and edit. One difference between a script and a command line is that the script may employ tab characters to achieve indentation, whereas the command line cannot, since tabs are used to activate completion.

Configuring For Script Writing

The text editor has many, many configuration settings. There are several common options that can facilitate script writing:

2 !

turns on syntax highlighting. With this setting, different elements of shell syntax will be displayed in different colors when viewing a script. This is helpful for identifying certain kinds of programming errors. It looks cool, too. Note that for this feature to work, you must have a complete version of installed, and the file you are editing must have a shebang indicating the file is a shell script. If you have difficulty with the command above, try 2 ! $ instead.

2

turns on the option to highlight search results. Say we search for the word “echo.” With this option on, each instance of the word will be highlighted.

2 $A

sets the number of columns occupied by a tab character. The default is 8 columns. Setting the value to 4 (which is a common practice) allows long lines to fit more easily on the screen.

2

turns on the “auto indent” feature. This causes to indent a new line the same amount as the line just typed. This speeds up typing on many kinds of programming constructs. To stop indentation, type <

These changes can be made permanent by adding these commands (without the leading colon characters) to your 5< file.

Summing Up

In this first chapter of scripting, we have looked at how scripts are written and made to easily execute on our system. We also saw how we may use various formatting techniques to improve the readability (and thus, the maintainability) of our scripts. In future chapters, ease of maintenance will come up again and again as a central principle in good script writing.

Further Reading

For “Hello World” programs and examples in various programming languages, see:
http://en.wikipedia.org/wiki/Hello_world

This Wikipedia article talks more about the shebang mechanism:
http://en.wikipedia.org/wiki/Shebang_(Unix)

* (2 9

Starting with this chapter, we will begin to build a program. The purpose of this project is to see how various shell features are used to create programs and, more importantly, create programs.

The program we will write is a . It will present various statistics about our system and its status, and will produce this report in HTML format, so we can view it with a web browser such as Firefox or Chrome.

Programs are usually built up in a series of stages, with each stage adding features and capabilities. The first stage of our program will produce a very minimal HTML document that contains no system information. That will come later.

First Stage: Minimal Document

The first thing we need to know is the format of a well-formed HTML document. It looks like this:

```
XR)-=C
    XR'39C
        X)@)='C:   )   X5)@)='C
    X5R'39C
    XGS9UC
        :     ,<
    X5GS9UC
X5R)-=C
```

If we enter this into our text editor and save the file as < , we can use the following URL in Firefox to view the file:

555 5 5 <

The first stage of our program will be able to output this HTML file to standard output. We can write a program to do this pretty easily. Let's start our text editor and create a new file named 5 5 , N N6 :

```
                              ! .   .
```

and enter the following program:

```
UL

U B                !

   J/R+OT&J
   J          /RG:1&J
   J                  /+P+TG&B    +    / +P+TG&J
   J          / RG:1&J
   J          /<S1^&J
   J                  B      ! J
   J          / <S1^&J
   J/ R+OT&J
```

Our first attempt at this problem contains a shebang, a comment (always a good idea) and a sequence of commands, one for each line of output. After saving the file, we'll make it executable and attempt to run it:

```
                              C""       ! .   .
                          ! .   .
```

When the program runs, we should see the text of the HTML document displayed on the screen, since the commands in the script send their output to standard output. We'll run the program again and redirect the output of the program to the file , N N6 < , so that we can view the result with a web browser:

```
                          ! .   .   & ! .   .
                              ! .   .
```

So far, so good.

When writing programs, it's always a good idea to strive for simplicity and clarity. Maintenance is easier when a program is easy to read and understand, not to mention that it can make the program easier to write by reducing the amount of typing. Our current version of the program works fine, but it could be simpler. We could actually combine all the echo commands into one, which will certainly make it easier to add more lines to the program's output. So, let's change our program to this:

```
ZF5  5

Z :         6     ,           6

   J/R+OT&
     /RG:1&
           /+P+TG&B    +    / +P+TG&
     / RG:1&
     /<S1^&
           B      !
     / <S1^&
/ R+OT&J
```

A quoted string may include newlines, and therefore contain multiple lines of text. The shell will keep reading the text until it encounters the closing quotation mark. It works this way on the command line, too:

```
                          J/R+OT&
C             /RG:1&
C                         /+P+TG&B    +    / +P+TG&
C             / RG:1&
C             /<S1^&
C                         B      !
C             / <S1^&
C  / R+OT&J
```

The leading “>” character is the shell prompt contained in the :(* shell variable. It appears whenever we type a multi-line statement into the shell. This feature is a little obscure right now, but later, when we cover multi-line programming statements, it will turn out to be quite handy.

Second Stage: Adding A Little Data

Now that our program can generate a minimal document, let’s put some data in the report. To do this, we will make the following changes:

```
ZF5  5

Z :         6     ,           6

   _XR)-=C
     XR'39C
          X)@)='C              F!    P         H                    X5)@)='C
```

```
    X5R'39C
    XGS9UC
        XR"C            F!   P       H                   X5R"C
    X5GS9UC
X5R)-=C_
```

We added a page title and a heading to the body of the report.

Variables And Constants

There is an issue with our script, however. Notice how the string “System Information Report” is repeated? With our tiny script it’s not a problem, but let’s imagine that our script was really long and we had multiple instances of this string. If we wanted to change the title to something else, we would have to change it in multiple places, which could be a lot of work. What if we could arrange the script so that the string only appeared once and not multiple times? That would make future maintenance of the script much easier. Here’s how we could do that:

```
ZF5  5

Z :       6    ,          6

   $JF!   P       H   J

   _XR)-=C
     XR'39C
        X)@)='C             3       X5)@)='C
     X5R'39C
     XGS9UC
        XR"C            3       X5R"C
     X5GS9UC
X5R)-=C_
```

By creating a ! named and assigning it the value “System Information Report”, we can take advantage of parameter expansion and place the string in multiple locations.

So, how do we create a variable? Simple, we just use it. When the shell encounters a variable, it automatically creates it. This differs from many programming languages in which variables must be explicitly or defined before use. The shell is very lax about this, which can lead to some problems. For example, consider this scenario played out on the command line:

```
                    $J! J
                     3
'
                     3
```

We first assign the value “yes” to the variable , and then display its value with . Next we display the value of the variable name misspelled as “fool” and get a blank result. This is because the shell happily created the variable when it encountered it, and gave it the default value of nothing, or empty. From this, we learn that we must pay close attention to our spelling! It’s also important to understand what really happened in this example. From our previous look at how the shell performs expansions, we know that the command:

```

```

undergoes parameter expansion and results in:

```
                  '
```

Whereas the command:

```

```

expands into:

```

```

The empty variable expands into nothing! This can play havoc with commands that require arguments. Here’s an example:

```
                    $
                     $
                    3  3
6             6      I < J
```

```
) , I 6    6J              <
```

We assign values to two variables, and " . We then perform a 6 , but misspell the name of the second argument. After expansion, the 6 command is only sent one argument, though it requires two.

There are some rules about variable names:

1. Variable names may consist of alphanumeric characters (letters and numbers) and underscore characters.
2. The first character of a variable name must be either a letter or an underscore.
3. Spaces and punctuation symbols are not allowed.

The word “variable” implies a value that changes, and in many applications, variables are used this way. However, the variable in our application, , is used as a . A constant is just like a variable in that it has a name and contains a value. The difference is that the value of a constant does not change. In an application that performs geometric calculations, we might define :@ as a constant, and assign it the value of /<"&"# , instead of using the number literally throughout our program. The shell makes no distinction between variables and constants; they are mostly for the programmer’s convenience. A common convention is to use uppercase letters to designate constants and lower case letters for true variables. We will modify our script to comply with this convention:

```
ZF5  5

Z :        6    ,          6

+P+TGL_(,   @       Q 6                  7  3RSF+?:OG  _

   _XR)-=C
     XR'39C
          X)@)='C              +P+TGX5)@)='C
     X5R'39C
     XGS9UC
          XR"C              +P+TGX5R"C
     X5GS9UC
X5R)-=C_
```

We also took the opportunity to jazz up our title by adding the value of the shell variable RS()K3-' . This is the network name of the machine.

2 : The shell actually does provide a way to enforce the immutability of constants, through the use of the builtin command with the (read-only) option. Had we assigned)@)=' this way:

)@)='L_:) _

the shell would prevent any subsequent assignment to)@)=' . This feature is rarely used, but it exists for very formal scripts.

Assigning Values To Variables And Constants

Here is where our knowledge of expansion really starts to pay off. As we have seen, variables are assigned values this way:

L

where ! is the name of the variable and is a string. Unlike some other programming languages, the shell does not care about the type of data assigned to a variable; it treats them all as strings. You can force the shell to restrict the assignment to integers by using the command with the option, but, like setting variables as read-only, this is rarely done.

Note that in an assignment, there must be no spaces between the variable name, the equals sign, and the value. So what can the value consist of? Anything that we can expand into a string:

```
LB                    Z 3          _B_        <
L_     _              Z '     6       8    A   <
L_          _         Z S     6
                      Z  6              <
L M      <  O         Z Q           <
L MM# D 0OO           Z  3       6    <
L_^ ^      ^ _        Z '  6   A              8   <
```

Multiple variable assignments may be done on a single line:

```
L#  L_     _
```

During expansion, variable names may be surrounded by optional curly braces “{}”. This is useful in cases where a variable name becomes ambiguous due to its surrounding con-

text. Here, we try to change the name of a file from , to , " , using a variable:

```
                    $J !   J
                   3
                 3      3
            6       I ,   J
) , I     6J          <
```

This attempt fails because the shell interprets the second argument of the command as a new (and empty) variable. The problem can be overcome this way:

```
                 3      39      >
```

By adding the surrounding braces, the shell no longer interprets the trailing " as part of the variable name.

We'll take this opportunity to add some data to our report, namely the date and time the report was created and the username of the creator:

```
ZF5   5

Z :          6     ,            6

)@)='L_(,     @          Q 6        RS()K3-'_
=EHHG?+.+POG$34     5J8  8  8@J6
+POGF+:OB$JQ        3=EHHG?+.+POG;  ! 3EFGHJ

    _XR)-=C
      XR'39C
            X)@)='C )@)='X5)@)='C
      X5R'39C
      XGS9UC
            XR"C )@)='X5R"C
                        /B&3+POGF+:OB/ B&
      X5GS9UC
X5R)-=C_
```

Here Documents

We've looked at two different methods of outputting our text, both using the com-

mand. There is a third way called a or . A here document is an additional form of I/O redirection in which we embed a body of text into our script and feed it into the standard input of a command. It works like this:

XX

where is the name of command that accepts standard input and is a string used to indicate the end of the embedded text. We'll modify our script to use a here document:

```
ZF5   5

Z :          6     ,             6

)@)='L_(,     @          Q 6        RS()K3-'_
 2QQ'K)N)@-'L M     7_4  4  4H_O
)@-'()3-:L_?          2QQ'K)N)@-'>  ,  2('Q_

   // .GS7.
XR)-=C
       XR'39C
            X)@)='C )@)='X5)@)='C
       X5R'39C
       XGS9UC
            XR"C )@)='X5R"C
            X:C )@-'()3-:X5:C
       X5GS9UC
X5R)-=C
.GS7.
```

Instead of using , our script now uses and a here document. The string N'S N (meaning "End Of File," a common convention) was selected as the token, and marks the end of the embedded text. Note that the token must appear alone and that there must not be trailing spaces on the line.

So what's the advantage of using a here document? It's mostly the same as , except that, by default, single and double quotes within here documents lose their special meaning to the shell. Here is a command line example:

```
                        $J       J
                         // .GS7.
C  3
```

```
C  J3   J
C  %3   %
C  K3
C  .GS7.


J       J
```

As we can see, the shell pays no attention to the quotation marks. It treats them as ordinary characters. This allows us to embed quotes freely within a here document. This could turn out to be handy for our report program.

Here documents can be used with any command that accepts standard input. In this example, we use a here document to pass a series of commands to the 6 program in order to retrieve a file from a remote FTP server:

```
ZF5   5

Z (   6                    ):

 ):N('Q;'QL  6<  <     <
 ):N:3)RL5     5    5    ,5    5         /!+5      5     5
Q'-S)'N @='L        N    <   < B

  6    XX N'S N
 6     ):N('Q;'Q
         ,
     ):N:3)R

     Q'-S)'N @='
 ,
N'S N
      Q'-S)'N @='
```

If we change the redirection operator from “<<” to “<<-”, the shell will ignore leading tab characters in the here document. This allows a here document to be indented, which can improve readability:

```
ZF5   5

Z (   6                    ):

 ):N('Q;'QL  6<  <     <
```

```
 ):N:3)RL5     5    5   ,5   5        /!+5      5     5
Q'-S)'N @='L       N   <   < B

 6    XX  N'S N
       6    ):N('Q;'Q
              ,
           ):N:3)R

           Q'-S)'N @='
         ,
        N'S N

       Q'-S)'N @='
```

Summing Up

In this chapter, we started a project that will carry us through the process of building a successful script. We introduced the concept of variables and constants and how they can be employed. They are the first of many applications we will find for parameter expansion. We also looked at how to produce output from our script, and various methods for embedding blocks of text.

Further Reading

For more information about HTML, see the following articles and tutorials:
http://en.wikipedia.org/wiki/Html
http://en.wikibooks.org/wiki/HTML_Programming
http://html.net/tutorials/html/

The man page includes a section entitled “HERE DOCUMENTS,” which has a full description of this feature.

, ":) 5)

As programs get larger and more complex, they become more difficult to design, code and maintain. As with any large project, it is often a good idea to break large, complex tasks into a series of small, simple tasks. Let's imagine that we are trying to describe a common, everyday task, going to the market to buy food, to a person from Mars. We might describe the overall process as the following series of steps:

1. Get in car.
2. Drive to market.
3. Park car.
4. Enter market.
5. Purchase food.
6. Return to car.
7. Drive home.
8. Park car.
9. Enter house.

However, a person from Mars is likely to need more detail. We could further break down the subtask "Park car" into this series of steps:

1. Find parking space.
2. Drive car into space.
3. Turn off motor.
4. Set parking brake.
5. Exit car.
6. Lock car.

The "Turn off motor" subtask could further be broken down into steps including "Turn off ignition," "Remove ignition key," and so on, until every step of the entire process of going to the market has been fully defined.

This process of identifying the top-level steps and developing increasingly detailed views of those steps is called . This technique allows us to break large complex tasks into many small, simple tasks. Top-down design is a common method of designing

programs and one that is well suited to shell programming in particular.

In this chapter, we will use top-down design to further develop our report-generator script.

Shell Functions

Our script currently performs the following steps to generate the HTML document:

1. Open page.
2. Open page header.
3. Set page title.
4. Close page header.
5. Open page body.
6. Output page heading.
7. Output timestamp.
8. Close page body.
9. Close page.

For our next stage of development, we will add some tasks between steps 7 and 8. These will include:

> System uptime and load. This is the amount of time since the last shutdown or reboot and the average number of tasks currently running on the processor over several time intervals.
>
> Disk space. The overall use of space on the system’s storage devices.
>
> Home space. The amount of storage space being used by each user.

If we had a command for each of these tasks, we could add them to our script simply through command substitution:

```
ZF5   5

Z :          6     ,             6

)@)='L_(,     @          Q 6        RS()K3-'_
 2QQ'K)N)@-'L M     7_4  4  4H_O
)@-'()3-:L_?          2QQ'K)N)@-'>  ,  2('Q_

   XX N'S N
XR)-=C
      XR'39C
```

```
          X)@)='C )@)='X5)@)='C
     X5R'39C
     XGS9UC
          XR"C )@)='X5R"C
          X:C )@-'()3-:X5:C
                    34    .    6
          34    .   .   6
          34    .   .   6
     X5GS9UC
X5R)-=C
N'S N
```

We could create these additional commands two ways. We could write three separate scripts and place them in a directory listed in our PATH, or we could embed the scripts within our program as . As we have mentioned before, shell functions are "mini-scripts" that are located inside other scripts and can act as autonomous programs. Shell functions have two syntactic forms. First, the more formal form:

\

]

and the simpler (and generally preferred) form:

MO \

]

where is the name of the function and is a series of commands contained within the function. Both forms are equivalent and may be used interchangeably. Below we see a script that demonstrates the use of a shell function:

```
"    ZF5  5
*
/    Z (
&
#          6* \
+            _(  6 *_
0
!    ]
%
"$   Z -   6
""
"*       _(  6 "_
```

```
"/       6*
"&        _(  6 /_
```

As the shell reads the script, it passes over lines 1 through 11, as those lines consist of comments and the function definition. Execution begins at line 12, with an com-mand. Line 13 the shell function 6* and the shell executes the function just as it would any other command. Program control then moves to line 6, and the second command is executed. Line 7 is executed next. Its command terminates the function and returns control to the program at the line following the function call (line 14), and the final command is executed. Note that in order for function calls to be recognized as shell functions and not interpreted as the names of external programs, shell function definitions must appear in the script before they are called.

We'll add minimal shell function definitions to our script:

```
ZF5   5

Z :          6     ,             6

)@)='L_(,     @          Q 6        RS()K3-'_
 2QQ'K)N)@-'L M     7_4  4  4H_O
)@-'()3-:L_?          2QQ'K)N)@-'>  ,  2('Q_

    .      46 9

>

    .   .     46 9

>

    .   .     46 9

>

   XX N'S N
XR)-=C
      XR'39C
            X)@)='C )@)='X5)@)='C
      X5R'39C
      XGS9UC
            XR"C )@)='X5R"C
            X:C )@-'()3-:X5:C
             M  6   N 6    O
             M  6   N    N 6   O
             M  6   N    N 6   O
```

```
      X5GS9UC
X5R)-=C
N'S N
```

Shell function names follow the same rules as variables. A function must contain at least one command. The command (which is optional) satisfies the requirement.

Local Variables

In the scripts we have written so far, all the variables (including constants) have been ! ! . Global variables maintain their existence throughout the program. This is fine for many things, but it can sometimes complicate the use of shell functions. Inside shell functions, it is often desirable to have ! . Local variables are only accessible within the shell function in which they are defined and cease to exist once the shell function terminates.

Having local variables allows the programmer to use variables with names that may already exist, either in the script globally or in other shell functions, without having to worry about potential name conflicts.

Here is an example script that demonstrates how local variables are defined and used:

```
ZF5   5

Z            6

  L$         Z

    N" MO \

                    Z                    N"

        L"
          _    N"     L    _
]

    N* MO \

                    Z                    N*

        L*
          _    N*     L    _
]

    _          L    _
    N"
```

```
_          L    _
N*
_          L    _
```

As we can see, local variables are defined by preceding the variable name with the word . This creates a variable that is local to the shell function in which it is defined. Once outside the shell function, the variable no longer exists. When we run this script, we see the results:

```
       L $
N"      L "
       L $
N*      L *
       L $
```

We see that the assignment of values to the local variable within both shell functions has no effect on the value of defined outside the functions.

This feature allows shell functions to be written so that they remain independent of each other and of the script in which they appear. This is very valuable, as it helps prevent one part of a program from interfering with another. It also allows shell functions to be written so that they can be portable. That is, they may be cut and pasted from script to script, as needed.

Keep Scripts Running

While developing our program, it is useful to keep the program in a runnable state. By doing this, and testing frequently, we can detect errors early in the development process. This will make debugging problems much easier. For example, if we run the program, make a small change, then run the program again and find a problem, it's very likely that the most recent change is the source of the problem. By adding the empty functions, called ! in programmer-speak, we can verify the logical flow of our program at an early stage. When constructing a stub, it's a good idea to include something that provides feedback to the programmer, which shows the logical flow is being carried out. If we look at the output of our script now:

```
                          ! .    .
XR)-=C
      XR'39C
             X)@)='C(,     @          Q 6        8  *X5)@)='C
```

```
     X5R'39C
     XGS9UC
          XR"C(,    @         Q 6             X5R"C
          X:C?       $/5"%5*$$% $& $* "$ :- '9)>  ,   X5:C



     X5GS9UC
X5R)-=C
```

we see that there are some blank lines in our output after the timestamp, but we can't be sure of the cause. If we change the functions to include some feedback:

```
 6   N 6    MO \
             J7          .            J

]

 6   N    N 6    MO \
             J7          .   .          J

]

 6   N    N 6    MO \
             J7          .   .          J

]
```

and run the script again:

```
                        ! .   .
XR)-=C
     XR'39C
          X)@)='C(,    @         Q 6             X5)@)='C
     X5R'39C
     XGS9UC
          XR"C(,    @         Q 6             X5R"C
          X:C?       $/5*$5*$$% $# "0 *+ 3- '9)>  ,   X5:C
                6   N 6          <
                6   N    N 6        <
                6   N    N 6        <
     X5GS9UC
X5R)-=C
```

we now see that, in fact, our three functions are being executed.

With our function framework in place and working, it's time to flesh out some of the function code. First, the 6 N 6 function:

```
 6  N 6    MO \
       //  .GS7.
           /R &F!    E    / R &
           /BHG&34    6/ BHG&
           .GS7.

]
```

It's pretty straightforward. We use a here document to output a section header and the output of the 6 command, surrounded by X:Q'C tags to preserve the formatting of the command. The 6 N N 6 function is similar:

```
 6  N   N 6   MO \
       //  .GS7.
           /R &1   F    E   -    / R &
           /BHG&34    6/ BHG&
           .GS7.

]
```

This function uses the command to determine the amount of disk space. Lastly, we'll build the 6 N N 6 function:

```
 6  N   N 6   MO \
       //  .GS7.
           /R &R   F    E   -    / R &
           /BHG&34         6/ BHG&
           .GS7.

]
```

We use the command with the options to perform this task. This, however, is not a complete solution to the problem. While it will work on some systems (Ubuntu, for example), it will not work on others. The reason is that many systems set the permissions of home directories to prevent them from being world-readable, which is a reasonable security measure. On these systems, the 6 N N 6 function, as written, will

only work if our script is run with superuser privileges. A better solution would be to have the script adjust its behavior according to the privileges of the user. We will take this up in the next chapter.

Shell Functions In Your File

Shell functions make excellent replacements for aliases, and are actually the preferred method of creating small commands for personal use. Aliases are very limited in the kind of commands and shell features they support, whereas shell functions allow anything that can be scripted. For example, if we liked the 6 N N 6 shell function that we developed for our script, we could create a similar function named for our . file:

```
MO \
      c9   (6   2   B       RS()K3-'d

]
```

Summing Up

In this chapter, we have introduced a common method of program design called top-down design, and we have seen how shell functions are used to build the stepwise refinement that it requires. We have also seen how local variables can be used to make shell functions independent from one another and from the program in which they are placed. This makes it possible for shell functions to be written in a portable manner and to be
! by allowing them to be placed in multiple programs; a great time saver.

Further Reading

The Wikipedia has many articles on software design philosophy. Here are a couple of good ones:
http://en.wikipedia.org/wiki/Top-down_design
http://en.wikipedia.org/wiki/Subroutines

' 5 ; 6

In the last chapter, we were presented with a problem. How can we make our report-generator script adapt to the privileges of the user running the script? The solution to this problem will require us to find a way to “change directions” within our script, based on the results of a test. In programming terms, we need the program to ! .

Let’s consider a simple example of logic expressed in , a simulation of a computer language intended for human consumption:

X = 5

If X = 5, then:

Say “X equals 5.”

Otherwise:

Say “X is not equal to 5.”

This is an example of a branch. Based on the condition, “Does X = 5?” do one thing, “Say X equals 5,” otherwise do another thing, “Say X is not equal to 5.”

Using the shell, we can code the logic above as follows:

```
L#

      A # T
      _  A   #<_

      _       A   #<_
```

or we can enter it directly at the command line (slightly shortened):

```
                    $"
                     ( 3  ," )#      J ,  "J#
J     ,  "J#
 A   #
                    $*
                     ( 3  ," )#      J ,  "J#
J     ,  "J#
     A  #
```

In this example, we execute the command twice. Once, with the value of set to 5, which results in the string “equals 5” being output, and the second time with the value of set to 0, which results in the string “does not equal 5” being output.

The statement has the following syntax:

```
          T

            T
           <<<
```

where is a list of commands. This is a little confusing at first glance. But before we can clear this up, we have to look at how the shell evaluates the success or failure of a command.

Exit Status

Commands (including the scripts and shell functions we write) issue a value to the system when they terminate, called an . This value, which is an integer in the range of 0 to 255, indicates the success or failure of the command’s execution. By convention, a value of zero indicates success and any other value indicates failure. The shell provides a parameter that we can use to examine the exit status. Here we see it in action:

```
5  5
                    3_
$

        5  5   K            ,
                    3_
*
```

In this example, we execute the command twice. The first time, the command executes successfully. If we display the value of the parameter E , we see that it is zero. We execute the command a second time (specifying a non-existent directory) , producing an error, and examine the parameter E again. This time it contains a 2, indicating that the command encountered an error. Some commands use different exit status values to provide diagnostics for errors, while many commands simply exit with a value of one when they fail. Man pages often include a section entitled “Exit Status,” describing what codes are used. However, a zero always indicates success.

The shell provides two extremely simple builtin commands that do nothing except terminate with either a zero or one exit status. The command always executes successfully and the command always executes unsuccessfully:

```
                    3_
$
                    3_
"
```

We can use these commands to see how the statement works. What the statement really does is evaluate the success or failure of commands:

```
                     #      JP %    J#
@ J    <
                     #      JP %    J#
```

The command _@ J <_ is executed when the command following executes successfully, and is not executed when the command following does not execute successfully. If a list of commands follows , the last command in the list is evaluated:

```
                    T   T      _@ J   <_T
@ J    <
                    T   T      _@ J   <_T
```

By far, the command used most frequently with is . The command performs a variety of checks and comparisons. It has two equivalent forms:

and the more popular:

where is an expression that is evaluated as either true or false. The command returns an exit status of zero when the expression is true and a status of one when the expression is false.

It is interesting to note that both and are actually commands. In they are builtins, but they also exist as programs in 5 5 for use with other shells. The expression is actually just its arguments with the command requiring that the “]” character be provided as its final argument.

The and commands support a wide range of useful expressions and tests.

File Expressions

The following expressions are used to evaluate the status of files:

! "4 1

Expression	Is True If:
	and " have the same inode numbers (the two filenames refer to the same file by hard linking).
	is newer than " .
	is older than " .
	exists and is a block-special (device) file.
	exists and is a character-special (device) file.
	exists and is a directory.
	exists.
	exists and is a regular file.
	exists and is set-group-ID.
?	exists and is owned by the effective group ID.
	exists and has its “sticky bit” set.

=	exists and is a symbolic link.
S	exists and is owned by the effective user ID.
6	exists and is a named pipe.
	exists and is readable (has readable permission for the effective user).
	exists and has a length greater than zero.
(	exists and is a network socket.
	is a file descriptor directed to/from the terminal. This can be used to determine whether standard input/output/error is being redirected.
	exists and is setuid.
8	exists and is writable (has write permission for the effective user).
	exists and is executable (has execute/search permission for the effective user).

Here we have a script that demonstrates some of the file expressions:

```
ZF5  5

Z       '

 @='L 5<

    _ @='_ T
        _ @='_ T
            _ @='          <_

        _ @='_ T
            _ @='        ,<_

        _ @='_ T
            _ @='       <_

       8 _ @='_ T
            _ @='  8    <_

        _ @='_ T
            _ @='        5      <_
```

```
_ @='          _
"
```

The script evaluates the file assigned to the constant @=' and displays its results as the evaluation is performed. There are two interesting things to note about this script. First, notice how the parameter @=' is quoted within the expressions. This is not required to syntacticly complete the expression, rather it is a defense against the parameter being empty. If the parameter expansion of @=' were to result in an empty value, it would cause an error (the operators would be interpreted as non-null strings rather than operators). Using the quotes around the parameter ensures that the operator is always followed by a string, even if the string is empty. Second, notice the presence of the command near the end of the script. The command accepts a single, optional argument, which becomes the script's exit status. When no argument is passed, the exit status defaults to the exit status of the last command executed. Using in this way allows the script to indicate failure if @=' expands to the name of a nonexistent file. The command appearing on the last line of the script is there as a formality. When a script "runs off the end" (reaches end of file), it terminates with an exit status of the last command executed.

Similarly, shell functions can return an exit status by including an integer argument to the command. If we were to convert the script above to a shell function to include it in a larger program, we could replace the commands with statements and get the desired behavior:

```
N    MO \

  Z        '

   @='L 5<

      _ @='_ T
           _ @='_ T
                _ @='          <_

           _ @='_ T
                _ @='        ,<_

           _ @='_ T
```

```
                    _ @='        <_

               8 _  @='_ T
                    _ @='   8     <_

                _ @='_ T
                    _ @='         5       <_


               _ @='          _
                 "


]
```

String Expressions

The following expressions are used to evaluate strings:

! "4 " # 1

Expression	Is True If...
	is not null.
	The length of is greater than zero.
B	The length of is zero.
L !!	and " are equal. Single or double equal signs may be used, but the use of double equal signs is greatly preferred.
FL	and " are not equal.
C	sorts after " .
X	sorts before " .

' : the C and X expression operators must be quoted (or escaped with a backslash) when used with . If they are not, they will be interpreted by the shell as redirection operators, with potentially destructive results. Also note that while the documentation states that the sorting order conforms to the collation order of the current locale, it does not. ASCII (POSIX) order is used in versions of up to and including 4.0.

Here is a script that incorporates string expressions:

```
ZF5  5

Z

3K(.'QL  ,

    B _ 3K(.'Q_  T
        _)          8  <_ CY*
         "


   _ 3K(.'Q_ L _,  _  T
        _)     8     U'(<_
    _ 3K(.'Q_ L _  _  T
        _)     8     KS<_
    _ 3K(.'Q_ L _  ,  _  T
        _)     8     -3UG'<_

        _)     8     2K1KS.K<_
```

In this script, we evaluate the constant 3K(.'Q . We first determine if the string is empty. If it is, we terminate the script and set the exit status to one. Notice the redirection that is applied to the echo command. This redirects the error message “There is no answer.” to standard error, which is the proper thing to do with error messages. If the string is not empty, we evaluate the value of the string to see if it is equal to either “yes,” “no,” or “maybe.” We do this by using , which is short for “else if.” By using , we are able to construct a more complex logical test.

Integer Expressions

The following expressions are used with integers:

! "4 (+ 1

Expression	Is True If...
A	is equal to " .
	is not equal to " .
	is less than or equal to " .
	is less than " .

	is greater than or equal to " .
	is greater than " .

Here is a script that demonstrates them:

```
ZF5  5

Z                              <

@K)L #

   B _ @K)_  T
       _@K)     6 ,<_ CY*
       "


   @K)  A $  T
       _@K)   B  <_

        @K)    $  T
              _@K)          <_

              _@K)   6      <_

        MM@K) 4 *OO   A $  T
              _@K)       <_

              _@K)      <_
```

The interesting part of the script is how it determines whether an integer is even or odd. By performing a modulo 2 operation on the number, which divides the number by two and returns the remainder, it can tell if the number is odd or even.

A More Modern Version Of

Recent versions of include a compound command that acts as an enhanced replacement for . It uses the following syntax:

where, like , is an expression that evaluates to either a true or false result. The command is very similar to (it supports all of its expressions), but

adds an important new string expression:

L

which returns true if is matched by the extended regular expression . This opens up a lot of possibilities for performing such tasks as data validation. In our earlier example of the integer expressions, the script would fail if the constant @K) contained anything except an integer. The script needs a way to verify that the constant contains an integer. Using with the L string expression operator, we could improve the script this way:

```
ZF5  5

Z        *                    <

@K)L #

  (( J3P?+J $  [ _(* D)53 ))#
        @K)  A $  T
             _@K)   B  <_

             @K)    $  T
                  _@K)        <_

                  _@K)   6     <_

             MM@K) 4 *OO   A $  T
                  _@K)      <_

                  _@K)     <_



       JP?+             J &'
```

By applying the regular expression, we are able to limit the value of @K) to only strings that begin with an optional minus sign, followed by one or more numerals. This expression also eliminates the possibility of empty values.

Another added feature of is that the LL operator supports pattern matching the same way pathname expansion does. For example:

```
                    7PTG$
```

```
                        (( 37PTG $$      ))#
C     J37PTG            %    %J
C
  <         6     J   <DJ
```

This makes useful for evaluating file and pathnames.

44 66 - Designed For Integers

In addition to the compound command, also provides the MM OO compound command, which is useful for operating on integers. It supports a full set of arithmetic evaluations, a subject we will cover fully in Chapter 34.

MM OO is used to perform . An arithmetic truth test results in true if the result of the arithmetic evaluation is non-zero.

```
                        44 66#          JP        J#
@       <
                        44*66#          JP        J#
```

Using MM OO, we can slightly simplify the * script like this:

```
ZF5   5

Z          *                        <

@K)L #

    _ @K)_ L  ` E $ % 7    T
         44P?+ $$ *66    T
              _@K)    B   <_

               44P?+ / *66     T
                    _@K)          <_

                    _@K)    6      <_

               44 44P?+ 8  66 $$ *66        T
                    _@K)       <_

                    _@K)      <_
```

```
_@K)              <_ CY*
"
```

Notice that we use less-than and greater-than signs and that LL is used to test for equivalence. This is a more natural-looking syntax for working with integers. Notice too, that because the compound command MM OOis part of the shell syntax rather than an ordinary command, and it deals only with integers, it is able to recognize variables by name and does not require expansion to be performed. We'll discuss MM OOand the related arithmetic expansion further in Chapter 34.

Combining Expressions

It's also possible to combine expressions to create more complex evaluations. Expressions are combined by using logical operators. We saw these in Chapter 17, when we learned about the command. There are three logical operations for and . They are AND, OR and NOT. and use different operators to represent these operations :

! "4 -)

Operation	(( )) **and** 44 66
AND	YY
OR	PP
NOT F	F

Here's an example of an AND operation. The following script determines if an integer is within a range of values:

```
ZF5  5

Z       /                8
Z 6             <

-@KN;3=L"
-3VN;3=L"$$

@K)L#$
```

```
_ @K)_ L  ` E $ % 7   T
     @K)   -@KN;3= YY @K)    -3VN;3=   T
           _ @K)   8     -@KN;3=    -3VN;3=<_

           _ @K)           <_


     _@K)            <_ CY*
     "
```

In this script, we determine if the value of integer @K) lies between the values of -@KN;3= and -3VN;3= . This is performed by a single use of , which includes two expressions separated by the YY operator. We could have also coded this using :

```
      (  @K)    -@KN;3=           @K)    -3VN;3=         ) T
           _ @K)   8     -@KN;3=    -3VN;3=<_

           _ @K)           <_
```

The F negation operator reverses the outcome of an expression. It returns true if an expression is false, and it returns false if an expression is true. In the following script, we modify the logic of our evaluation to find values of @K) that are outside the specified range:

```
ZF5  5

Z        &
Z 6             <

-@KN;3=L"
-3VN;3=L"$$

@K)L#$

    _ @K)_ L  ` E $ % 7   T
         F M@K)    -@KN;3= YY @K)    -3VN;3=O   T
               _ @K)          -@KN;3=    -3VN;3=<_

               _ @K)         <_
```

```
_@K)              <_ CY*
"
```

We also include parentheses around the expression, for grouping. If these were not included, the negation would only apply to the first expression and not the combination of the two. Coding this with would be done this way:

```
F ^M  @K)     -@KN;3=    @K)     -3VN;3= ^O  T
      _ @K)          -@KN;3=    -3VN;3=<_

      _ @K)         <_
```

Since all expressions and operators used by are treated as command arguments by the shell (unlike and MM OO), characters which have special meaning to , such as X, C, M, and O, must be quoted or escaped.

Seeing that and do roughly the same thing, which is preferable? is traditional (and part of POSIX), whereas is specific to . It's important to know how to use , since it is very widely used, but is clearly more useful and is easier to code, so it is preferred for modern scripts.

Portability Is The Hobgoblin Of Little Minds

If you talk to "real" Unix people, you quickly discover that many of them don't like Linux very much. They regard it as impure and unclean. One tenet of Unix users is that everything should be "portable." This means that any script you write should be able to run, unchanged, on any Unix-like system.

Unix people have good reason to believe this. Having seen what proprietary extensions to commands and shells did to the Unix world before POSIX, they are naturally wary of the effect of Linux on their beloved OS.

But portability has a serious downside. It prevents progress. It requires that things are always done using "lowest common denominator" techniques. In the case of shell programming, it means making everything compatible with , the original Bourne shell.

This downside is the excuse that proprietary software vendors use to justify their proprietary extensions, only they call them “innovations.” But they are really just lock-in devices for their customers.

The GNU tools, such as , have no such restrictions. They encourage portability by supporting standards and by being universally available. You can install and the other GNU tools on almost any kind of system, even Windows, without cost. So feel free to use all the features of . It’s portable.

Control Operators: Another Way To Branch

provides two control operators that can perform branching. The YY (AND) and PP (OR) operators work like the logical operators in the compound command. This is the syntax:

YY

and

PP

It is important to understand the behavior of these. With the YY operator, is executed and " is executed if, , is successful. With the PP operator, is executed and " is executed if, , is unsuccessful.

In practical terms, it means that we can do something like this:

```
"
```

This will create a directory named 6 , and if it succeeds, the current working directory will be changed to 6 . The second command is attempted only if the command is successful. Likewise, a command like this:

```
(( )) 00
```

will test for the existence of the directory 6 , and only if the test fails, will the directory be created. This type of construct is very handy for handling errors in scripts, a subject we will discuss more in later chapters. For example, we could do this in a script:

```
    6  PP    "
```

If the script requires the directory 6 , and it does not exist, then the script will terminate with an exit status of one.

Summing Up

We started this chapter with a question. How could we make our , N N6 script detect if the user had permission to read all the home directories? With our knowledge of , we can solve the problem by adding this code to the 6 N N 6 function:

```
 6  N   N 6   MO \
      (( 34    6  , * ))#
              //  .GS7.
                  /R &R   F    E   -    4:  E   6/ R &
                  /BHG&34          6/ BHG&
                  .GS7.

              //  .GS7.
                  /R &R   F    E   -    43EFGH6/ R &
                  /BHG&34     3RSOG6/ BHG&
                  .GS7.


]
```

We evaluate the output of the command. With the option, outputs the numeric user ID number of the effective user. The superuser is always zero and every other user is a number greater than zero. Knowing this, we can construct two different here documents, one taking advantage of superuser privileges, and the other, restricted to the user's own home directory.

We are going to take a break from the , N N6 program, but don't worry. It will be back. In the meantime, we'll cover some topics that we'll need when we resume our work.

Further Reading

There are several sections of the man page that provide further detail on the topics covered in this chapter:

Lists (covers the control operators PP and YY)

Compound Commands (covers , MM OOand)

CONDITIONAL EXPRESSIONS

SHELL BUILTIN COMMANDS (covers)

Further, the Wikipedia has a good article on the concept of pseudocode:

http://en.wikipedia.org/wiki/Pseudocode

. - / # "

The scripts we have written so far lack a feature common in most computer programs—
. That is, the ability of the program to interact with the user. While many programs don't need to be interactive, some programs benefit from being able to accept input directly from the user. Take, for example, this script from the previous chapter:

```
ZF5  5

Z       *                  <

@K)L #

   _ @K)_ L  ` E $ % 7   T
        @K)  A $  T
             _@K)   B  <_

             @K)    $  T
                  _@K)        <_

                  _@K)   6     <_

             MM@K) 4 *OO   A $  T
                  _@K)      <_

                  _@K)     <_



       _@K)            <_ CY*
       "
```

Each time we want to change the value of @K) we have to edit the script. It would be much more useful if the script could ask the user for a value. In this chapter, we will begin to look at how we can add interactivity to our programs.

– Read Values From Standard Input

The builtin command is used to read a single line of standard input. This command can be used to read keyboard input or, when redirection is employed, a line of data from a file. The command has the following syntax:

<<<

where is one or more of the available options listed below and ! is the name of one or more variables used to hold the input value. If no variable name is supplied, the shell variable Q':=U contains the line of data.

Basically, assigns fields from standard input to the specified variables. If we modify our integer evaluation script to use , it might look like this:

```
ZF5  5

Z                          <

    JB              & J


   _   _ L  ` E $ % 7   T
            A $  T
              _     B  <_

                  $  T
                   _          <_

                   _     6     <_

              MM   4 *OO   A $  T
                   _        <_

                   _        <_



        _@ 6                <_ CY*
        "
```

We use with the option (which suppresses the trailing newline on output) to display a prompt, and then use to input a value for the variable . Running this script results in this:

```
:            C                "
#  6    <
#    <
```

can assign input to multiple variables, as shown in this script:

```
ZF5  5

Z      6        6          ,

     _'              C _
     "  *  /  &  #

   _  " L J  "J_
   _  * L J  *J_
   _  / L J  /J_
   _  & L J  &J_
   _  # L J  #J_
```

In this script, we assign and display up to five values. Notice how behaves when given different numbers of values:

```
'            C
  " L J J
  * L J J
  / L J J
  & L J J
  # L J J

'            C
  " L J J
  * L JJ
  / L JJ
  & L JJ
  # L JJ

'            C
  " L J J
  * L J J
  / L J J
  & L J J
  # L J   J
```

If receives fewer than the expected number, the extra variables are empty, while an excessive amount of input results in the final variable containing all of the extra input.

If no variables are listed after the command, a shell variable, Q':=U , will be assigned all the input:

```
ZF5  5

Z               6

    _'              C _


  _Q':=U L J Q':=UJ_
```

Running this script results in this:

```

'               C
Q':=U L J     J
```

Options

supports the following options:

! "5)

Option	Description
	Assign the input to , starting with index zero. We will cover arrays in Chapter 35.
	The first character in the string is used to indicate end of input, rather than a newline character.
	Use Readline to handle input. This permits input editing in the same manner as the command line.
	Use as a default reply if the user simply presses Enter. Requires the option.
	Read characters of input, rather than an entire line.
6	Display a prompt for input using the string .

	Raw mode. Do not interpret backslash characters as escapes.
	Silent mode. Do not echo characters to the display as they are typed. This is useful when inputting passwords and other confidential information.
	Timeout. Terminate input after . returns a non-zero exit status if an input times out.
	Use input from file descriptor , rather than standard input.

Using the various options, we can do interesting things with . For example, with the 6 option, we can provide a prompt string:

```
ZF5  5

Z              6

   6 _'              C _

  _Q':=U L J Q':=UJ_
```

With the and options we can write a script that reads “secret” input and times out if the input is not completed in a specified time:

```
ZF5  5

Z         6       6  6

      "$  6 _'       6  6    C _     N6  T
        _^ (    6  6    L J     N6  J_

        _^ @ 6        _ CY*
       "
```

The script prompts the user for a secret passphrase and waits 10 seconds for input. If the entry is not completed within the specified time, the script exits with an error. Since the option is included, the characters of the passphrase are not echoed to the display as they are typed.

It's possible to supply the user with a default response using the and options together:

```
ZF5  5

Z            66 ,                6     '     ,<

     6 _.     ,          E _    2('Q
   _U    8     J Q':=UJ_
```

In this script, we prompt the user to enter his/her user name and use the environment variable 2('Q to provide a default value. When the script is run it displays the default string and if the user simply presses the Enter key, will assign the default string to the Q':=U variable.

```
.    ,        E
U    8    J  J
```

IFS

Normally, the shell performs word splitting on the input provided to . As we have seen, this means that multiple words separated by one or more spaces become separate items on the input line, and are assigned to separate variables by . This behavior is configured by a shell variable named @ ((for Internal Field Separator). The default value of @ (contains a space, a tab, and a newline character, each of which will separate items from one another.

We can adjust the value of @ (to control the separation of fields input to . For example, the 5 56 8 file contains lines of data that use the colon character as a field separator. By changing the value of @ (to a single colon, we can use to input the contents of 5 56 8 and successfully separate fields into different variables. Here we have a script that does just that:

```
ZF5  5

Z

 @='L5  56  8
```

```
 6 _'          C _    N

N   L M  6 _`    N    _  @='O

  _    N   _ T
  @ (L_ _       68              XXX _    N   _
     _2   L    J   J_
     _2@9 L     J   J_
     _?@9 L     J   J_
     _    K   L J   J_
     _R   9  < L J   J_
     _(   L    J    J_

     _K        J    N   J_ CY*
     "
```

This script prompts the user to enter the username of an account on the system, then displays the different fields found in the user's record in the 5 56 8 file. The script contains two interesting lines. The first is:

```
N   L M  6 _`    N    _  @='O
```

This line assigns the results of a 6 command to the variable N . The regular expression used by 6 assures that the username will only match a single line in the 5 56 8 file.

The second interesting line is this one:

```
@ (L_ _        68              XXX _    N   _
```

The line consists of three parts: a variable assignment, a command with a list of variable names as arguments, and a strange new redirection operator. We'll look at the variable assignment first.

The shell allows one or more variable assignments to take place immediately before a command. These assignments alter the environment for the command that follows. The effect of the assignment is temporary; only changing the environment for the duration of the command. In our case, the value of @ (is changed to a colon character. Alternately, we could have coded it this way:

```
S=9N@ (L_ @ (_
@ (L_ _
      68              XXX _    N   _
@ (L_ S=9N@ (_
```

where we store the value of @ (, assign a new value, perform the command, and then restore @ (to its original value. Clearly, placing the variable assignment in front of

the command is a more concise way of doing the same thing.

The XXX operator indicates a . A here string is like a here document, only shorter, consisting of a single string. In our example, the line of data from the 5 56 8 file is fed to the standard input of the command. We might wonder why this rather oblique method was chosen rather than:

```
_  N  _ P @ (L_ _      68
```

Well, there's a reason...

You Can't Pipe

While the command normally takes input from standard input, you cannot do this:

```
_  _ P
```

We would expect this to work, but it does not. The command will appear to succeed but the Q':=U variable will always be empty. Why is this?

The explanation has to do with the way the shell handles pipelines. In (and other shells such as), pipelines create ! . These are copies of the shell and its environment which are used to execute the command in the pipeline. In our example above, is executed in a subshell.

Subshells in Unix-like systems create copies of the environment for the processes to use while they execute. When the processes finishes the copy of the environment is destroyed. This means that ! . assigns variables, which then become part of the environment. In the example above, assigns the value "foo" to the variable Q' :=U in its subshell's environment, but when the command exits, the subshell and its environment are destroyed, and the effect of the assignment is lost.

Using here strings is one way to work around this behavior. Another method is discussed in Chapter 36.

Validating Input

With our new ability to have keyboard input comes an additional programming challenge, validating input. Very often the difference between a well-written program and a poorly written one lies in the program's ability to deal with the unexpected. Frequently, the unexpected appears in the form of bad input. We've done a little of this with our evaluation programs in the previous chapter, where we checked the values of integers and screened

out empty values and non-numeric characters. It is important to perform these kinds of programming checks every time a program receives input, to guard against invalid data. This is especially important for programs that are shared by multiple users. Omitting these safeguards in the interests of economy might be excused if a program is to be used once and only by the author to perform some special task. Even then, if the program performs dangerous tasks such as deleting files, it would be wise to include data validation, just in case.

Here we have an example program that validates various kinds of input:

```
ZF5   5

Z                 6

    N  6   MO \
       _@        6   J Q':=UJ_ CY*
       "
]

   6 _'              C _

Z  6       6 , M      O
  B  Q':=U    YY       N  6

Z  6         6        M      O
MM  M      Q':=U P 8   8O C " OO YY       N  6

Z     6              E
    Q':=U L  `         ^<N 7    T
       _J Q':=UJ                <_
          Q':=U   T
             _3       J Q':=UJ      <_

             _R 8    >     J Q':=UJ           <_


    Z     6           6         E
         Q':=U L  ` E         D^<         7    T
              _J Q':=UJ            6         <_

              _J Q':=UJ              6         <_


    Z     6           E
         Q':=U L  ` E         7    T
              _J Q':=UJ           <_

              _J Q':=UJ             <_
```

```
_)        J Q':=UJ                <_
```

This script prompts the user to enter an item. The item is subsequently analyzed to determine its contents. As we can see, the script makes use of many of the concepts that we have covered thus far, including shell functions, , MM OQ, the control operator YY, and , as well as a healthy dose of regular expressions.

Menus

A common type of interactivity is called . In menu-driven programs, the user is presented with a list of choices and is asked to choose one. For example, we could imagine a program that presented the following:

```
:    (

"< 9  6  , (,    @
*< 9  6  , 9   (6
/< 9  6  , R   (6   2   B
$< [

'            $ /  C
```

Using what we learned from writing our , N N6 program, we can construct a menu-driven program to perform the tasks on the above menu:

```
ZF5   5

Z                  ,            6



   _
:    (

"< 9  6  , (,    @
*< 9  6  , 9   (6
/< 9  6  , R   (6   2   B
$< [
_
    6 _'            $ /  C _
```

```
Q':=U L  ` $ /    T
     Q':=U LL $   T
          _:            <_


     Q':=U LL "   T
          _R        RS()K3-'_
       6


     Q':=U LL *   T



     Q':=U LL /   T
          M    O   A $   T
               _R   (6   2   B     M3   2    O_
                 5   5D

               _R   (6   2   B     M 2('QO_
                 RS-'




    _@         ,<_ CY*
    "
```

This script is logically divided into two parts. The first part displays the menu and inputs the response from the user. The second part identifies the response and carries out the selected action. Notice the use of the command in this script. It is used here to prevent the script from executing unnecessary code after an action has been carried out. The presence of multiple exit points in a program is generally a bad idea (it makes program logic harder to understand), but it works in this script.

Summing Up

In this chapter, we took our first steps toward interactivity; allowing users to input data into our programs via the keyboard. Using the techniques presented thus far, it is possible to write many useful programs, such as specialized calculation programs and easy-to-use front-ends for arcane command line tools. In the next chapter, we will build on the menu-driven program concept to make it even better.

Extra Credit

It is important to study the programs in this chapter carefully and have a complete understanding of the way they are logically structured, as the programs to come will be increasingly complex. As an exercise, rewrite the programs in this chapter using the command rather than the compound command. Hint: Use 6 to evaluate the regular expressions and evaluate the exit status. This will be good practice.

Further Reading

The 9 : % contains a chapter on builtins, which includes the command:

http://www.gnu.org/software/bash/manual/bashref.html#Bash-Builtins

1 ' 5 ;< " 5 =

In the previous chapter, we developed a menu-driven program to produce various kinds of system information. The program works, but it still has a significant usability problem. It only executes a single choice and then terminates. Even worse, if an invalid selection is made, the program terminates with an error, without giving the user an opportunity to try again. It would be better if we could somehow construct the program so that it could repeat the menu display and selection over and over, until the user chooses to exit the program.

In this chapter, we will look at a programming concept called , which can be used to make portions of programs repeat. The shell provides three compound commands for looping. We will look at two of them in this chapter, and the third in a later one.

Looping

Daily life is full of repeated activities. Going to work each day, walking the dog, slicing a carrot are all tasks that involve repeating a series of steps. Let's consider slicing a carrot. If we express this activity in pseudocode, it might look something like this:

1. get cutting board
2. get knife
3. place carrot on cutting board
4. lift knife
5. advance carrot
6. slice carrot
7. if entire carrot sliced, then quit, else go to step 4

Steps 4 through 7 form a . The actions within the loop are repeated until the condition, "entire carrot sliced," is reached.

8

can express a similar idea. Let's say we wanted to display five numbers in sequen-

tial order from one to five. a script could be constructed as follows:

```
ZF5  5

Z 8          6 ,

   L"

8            #  T

        L MM     7 "OO

   _      <_
```

When executed, this script displays the following:

```
"
*
/
&
#
     <
```

The syntax of the 8 command is:

8 T T

Like , 8 evaluates the exit status of a list of commands. As long as the exit status is zero, it performs the commands inside the loop. In the script above, the variable is created and assigned an initial value of 1. The 8 command evaluates the exit status of the compound command. As long as the command returns an exit status of zero, the commands within the loop are executed. At the end of each cycle, the command is repeated. After five iterations of the loop, the value of has increased to 6, the command no longer returns an exit status of zero and the loop terminates. The program continues with the next statement following the loop.

We can use a to improve the program from the previous chapter:

```
ZF5  5

Z 8                ,          6
```

```
1GT:^$    Z K                  6  ,

    (( 3HGBT^ L$ * ))#

        XX  N'S N
            :     (

            "< 9  6  , (,     @
            *< 9  6  , 9    (6
            /< 9  6  , R    (6    2    B
            $< [

     N'S N
         6 _'             $ /  C _

          Q':=U L  ` $ /     T
                Q':=U LL "   T
                      _R         RS()K3-'_
                   6
                      31GT:^

                Q':=U LL *   T

                      31GT:^

                Q':=U LL /   T
                      M     O   A $   T
                            _R    (6    2    B      M3   2    O_
                              5    5D

                            _R    (6    2    B      M 2('QO_
                              RS-'

                      31GT:^


               _@          ,<_
                31GT:^


   _:             <_
```

By enclosing the menu in a while loop, we are able to have the program repeat the menu display after each selection. The loop continues as long as Q':=U is not equal to “0” and the menu is displayed again, giving the user the opportunity to make another selection. At the end of each action, a 6 command is executed so the program will pause for a few seconds to allow the results of the selection to be seen before the screen is cleared and the menu is redisplayed. Once Q':=U is equal to “0,” indicating the “quit” selection,

the loop terminates and execution continues with the line following .

Breaking Out Of A Loop

provides two builtin commands that can be used to control program flow inside loops. The command immediately terminates a loop, and program control resumes with the next statement following the loop. The command causes the remainder of the loop to be skipped, and program control resumes with the next iteration of the loop. Here we see a version of the 8 program incorporating both and :

```
ZF5  5

Z 8      *          ,          6

9'=3UL/ Z K              6 ,

     #

      XX  N'S N
          :    (

          "< 9  6  , (,    @
          *< 9  6  , 9   (6
          /< 9  6  , R   (6   2   B
          $< [

    N'S N
        6 _'          $ /  C _

         Q':=U L  ` $ /    T
               Q':=U LL "   T
                    _R        RS()K3-'_
                  6
                    6  9'=3U


               Q':=U LL *   T

                    6  9'=3U


               Q':=U LL /   T
                     M    O   A $   T
                          _R   (6   2   B     M3   2   O_
                            5   5D
```

```
                    _R   (6   2   B     M 2('QO_
                      RS-'

                6  9'=3U


          (( 3HGBT^ $$ * ))#



           _@        ,<_
          6  9'=3U


  _:          <_
```

In this version of the script, we set up an (one that never terminates on its own) by using the command to supply an exit status to 8 . Since will always exit with a exit status of zero, the loop will never end. This is a surprisingly common scripting technique. Since the loop will never end on its own, it's up to the programmer to provide some way to break out of the loop when the time is right. In this script, the command is used to exit the loop when the "0" selection is chosen. The command has been included at the end of the other script choices to allow for more efficient execution. By using , the script will skip over code that is not needed when a selection is identified. For example, if the "1" selection is chosen and identified, there is no reason to test for the other selections.

The command is much like 8 , except instead of exiting a loop when a non-zero exit status is encountered, it does the opposite. An continues until it receives a zero exit status. In our 8 script, we continued the loop as long as the value of the variable was less than or equal to 5. We could get the same result by coding the script with :

```
ZF5   5

Z          6  ,

   L"

    (( 3       " ))                T
```

```
       L MM     7 "OO

  _     <_
```

By changing the test expression to # , will terminate the loop at the correct time. The decision of whether to use the 8 or loop is usually a matter of choosing the one that allows the clearest test to be written.

Reading Files With Loops

8 and can process standard input. This allows files to be processed with while and until loops. In the following example, we will display the contents of the < file used in earlier chapters:

```
ZF5  5

Z 8

8                T
    6    _9    4 ^ ;    4 ^ Q    4 ^ _ ^
               ^
               ^

     /
```

To redirect a file to the loop, we place the redirection operator after the statement. The loop will use to input the fields from the redirected file. The command will exit after each line is read, with a zero exit status until the end-of-file is reached. At that point, it will exit with a non-zero exit status, thereby terminating the loop. It is also possible to pipe standard input into a loop:

```
ZF5  5

Z 8      *

    ;          0                  8                T

    6    _9    4 ^ ;    4 ^ Q    4 ^ _ ^
               ^
               ^
```

Here we take the output of the command and display the stream of text. However, it is important to remember that since a pipe will execute the loop in a subshell, any variables created or assigned within the loop will be lost when the loop terminates.

Summing Up

With the introduction of loops, and our previous encounters with branching, subroutines and sequences, we have covered the major types of flow control used in programs. has some more tricks up its sleeve, but they are refinements on these basic concepts.

Further Reading

The 9 ? 9 from the Linux Documentation Project has some more examples of while loops:
http://tldp.org/LDP/Bash-Beginners-Guide/html/sect_09_02.html

The Wikipedia has an article on loops, which is part of a larger article on flow control:
http://en.wikipedia.org/wiki/Control_flow#Loops

0

Now that our scripts become more complex, it’s time to take a look at what happens when things go wrong. In this chapter, we’ll look at some of the common kinds of errors that occur in scripts, and examine a few useful techniques that can be used to track down and eradicate problems.

Syntactic Errors

One general class of errors is . Syntactic errors involve mistyping some element of shell syntax. In most cases, the shell will refuse to execute a script containing this type of error.

In the following discussions, we will use this script to demonstrate common types of errors:

```
ZF5  5

Z        6

   L"

       L " T
       _K      A    "<_

       _K        A    "<_
```

As written, this script runs successfully:

```

K       A    "<
```

Missing Quotes

If we edit our script and remove the trailing quote from the argument following the first command:

```
ZF5  5

Z        6

   L"

       L "  T
       J?       ,

       _K         A     "<_
```

Watch what happens:

```


5   5 5  5        "$    6    'S  8
      I_J
5   5 5  5        "/   ,          6
```

It generates two errors. Interestingly, the line numbers reported by the error messages are not where the missing quote was removed, but rather much later in the program. If we follow the program after the missing quote, we can see why. will continue looking for the closing quote until it finds one, which it does, immediately after the second command. After that, becomes very confused. The syntax of the subsequent command is broken because the statement is now inside a quoted (but open) string.

In long scripts, this kind of error can be quite hard to find. Using an editor with syntax highlighting will help since, in most cases, it will display quoted strings in a distinctive manner from other kinds of shell syntax. If a complete version of is installed, syntax highlighting can be enabled by entering the command:

```
2 !
```

Missing Or Unexpected Tokens

Another common mistake is forgetting to complete a compound command, such as or 8 . Let's look at what happens if we remove the semicolon after the in the command:

```
ZF5  5

Z        6

   L"

 (3    $ )
      _K      A    "<_

      _K        A    "<_
```

The result is this:

```
5  5 5  5       %  ,                                    6
I  J
5  5 5  5       % I  J
```

Again, the error message points to an error that occurs later than the actual problem. What happens is really pretty interesting. As we recall, accepts a list of commands and evaluates the exit code of the last command in the list. In our program, we intend this list to consist of a single command, , a synonym for test. The command takes what follows it as a list of arguments; in our case, four arguments: , 1, L, and . With the semicolon removed, the word is added to the list of arguments, which is syntactically legal. The following command is legal, too. It's interpreted as another command in the list of commands that will evaluate for an exit code. The is encountered next, but it's out of place, since the shell recognizes it as a (a word that has special meaning to the shell) and not the name of a command, hence the error message.

Unanticipated Expansions

It's possible to have errors that only occur intermittently in a script. Sometimes the script will run fine and other times it will fail because of the results of an expansion. If we return our missing semicolon and change the value of to an empty variable, we

can demonstrate:

```
ZF5  5

Z        6

   $

       L "  T
       _K       A     "<_

       _K         A     "<_
```

Running the script with this change results in the output:

```
5   5 5  5         0    L                          , 6       6
 K          A     "<
```

We get this rather cryptic error message, followed by the output of the second command. The problem is the expansion of the variable within the command. When the command:

```
      L "
```

undergoes expansion with being empty, the result is this:

```
  L "
```

which is invalid and the error is generated. The L operator is a binary operator (it requires a value on each side), but the first value is missing, so the command expects a unary operator (such as B) instead. Further, since the failed (because of the error), the command receives a non-zero exit code and acts accordingly, and the second command is executed.

This problem can be corrected by adding quotes around the first argument in the command:

```
_     _ L "
```

Then when expansion occurs, the result will be this:

```
__ L "
```

which yields the correct number of arguments. In addition to empty strings, quotes should be used in cases where a value could expand into multi-word strings, as with filenames containing embedded spaces.

Logical Errors

Unlike syntactic errors, do not prevent a script from running. The script will run, but it will not produce the desired result, due to a problem with its logic. There are countless numbers of possible logical errors, but here are a few of the most common kinds found in scripts:

1.) (It's easy to incorrectly code an if/then/else and have the wrong logic carried out. Sometimes the logic will be reversed, or it will be incomplete.
2. .;"" 0 When coding loops that employ counters, it is possible to overlook that the loop may require that the counting start with zero, rather than one, for the count to conclude at the correct point. These kinds of errors result in either a loop "going off the end" by counting too far, or else missing the last iteration of the loop by terminating one iteration too soon.
3. - (Most logic errors result from a program encountering data or situations that were unforeseen by the programmer. As we have seen, this can also include unanticipated expansions, such as a filename that contains embedded spaces that expands into multiple command arguments rather than a single filename.

Defensive Programming

It is important to verify assumptions when programming. This means a careful evaluation of the exit status of programs and commands that are used by a script. Here is an example, based on a true story. An unfortunate system administrator wrote a script to perform a maintenance task on an important server. The script contained the following two lines of code:

```
   N
D
```

There is nothing intrinsically wrong with these two lines, as long as the directory named in the variable, N , exists. But what happens if it does not? In that case, the command fails and the script continues to the next line and deletes the files in the current working directory. Not the desired outcome at all! The hapless administrator destroyed an important part of the server because of this design decision.

Let's look at some ways this design could be improved. First, it might be wise to make the execution of contingent on the success of :

```
   N   YY   D
```

This way, if the command fails, the command is not carried out. This is better, but still leaves open the possibility that the variable, N , is unset or empty, which would result in the files in the user's home directory being deleted. This could also be avoided by checking to see that N actually contains the name of an existing directory:

```
     N     YY     N    YY   D
```

Often, it is best to include logic to terminate the script and report an error when an situation such as the one above occurs:

```
Z 9          ,   N
   F   _   N   _  T
       _K        , J   N   J_ CY*
       "

 F     N   T
       _       J   N   J_ CY*
       "

 F   DT
       _            <        _ CY*
       "
```

Here, we check both the name, to see that it is that of an existing directory, and the suc-

cess of the command. If either fails, a descriptive error message is sent to standard error and the script terminates with an exit status of one to indicate a failure.

Verifying Input

A general rule of good programming is that if a program accepts input, it must be able to deal with anything it receives. This usually means that input must be carefully screened, to ensure that only valid input is accepted for further processing. We saw an example of this in the previous chapter when we studied the command. One script contained the following test to verify a menu selection:

```
Q':=U L  ` $ /
```

This test is very specific. It will only return a zero exit status if the string entered by the user is a numeral in the range of zero to three. Nothing else will be accepted. Sometimes these kinds of tests can be very challenging to write, but the effort is necessary to produce a high quality script.

Design Is A Function Of Time

When I was a college student studying industrial design, a wise professor stated that the amount of design on a project was determined by the amount of time given to the designer. If you were given five minutes to design a device “that kills flies,” you designed a flyswatter. If you were given five months, you might come up with a laser-guided “anti-fly system” instead.

The same principle applies to programming. Sometimes a “quick-and-dirty” script will do if it’s only going to be used once and only used by the programmer. That kind of script is common and should be developed quickly to make the effort economical. Such scripts don’t need a lot of comments and defensive checks. On the other hand, if a script is intended for &that is, a script that will be used over and over for an important task or by multiple users, it needs much more careful development.

Testing

Testing is an important step in every kind of software development, including scripts. There is a saying in the open-source world, “release early, release often,” which reflects this fact. By releasing early and often, software gets more exposure to use and testing.

Experience has shown that bugs are much easier to find, and much less expensive to fix, if they are found early in the development cycle.

In Chapter 26, we saw how stubs can be used to verify program flow. From the earliest stages of script development, they are a valuable technique to check the progress of our work.

Let’s look at the file-deletion problem above and see how this could be coded for easy testing. Testing the original fragment of code would be dangerous, since its purpose is to delete files, but we could modify the code to make the test safe:

```
      N     T
         N   T
                D     U +GF+P?Q

            _         J   N   J_ CY*
            "


      _           , J   N   J_ CY*
      "

   U +GF+P?Q
```

Since the error conditions already output useful messages, we don't have to add any. The most important change is placing an command just before the command to allow the command and its expanded argument list to be displayed, rather than the command actually being executed. This change allows safe execution of the code. At the end of the code fragment, we place an command to conclude the test and prevent any other part of the script from being carried out. The need for this will vary according to the design of the script.

We also include some comments that act as “markers” for our test-related changes. These can be used to help find and remove the changes when testing is complete.

Test Cases

To perform useful testing, it's important to develop and apply good . This is done by carefully choosing input data or operating conditions that reflect and cases. In our code fragment (which is very simple), we want to know how the code performs under three specific conditions:

1. N contains the name of an existing directory
2. N contains the name of a non-existent directory

3. N is empty

By performing the test with each of these conditions, good is achieved.

Just as with design, testing is a function of time, as well. Not every script feature needs to be extensively tested. It's really a matter of determining what is most important. Since it could be so potentially destructive if it malfunctioned, our code fragment deserves careful consideration during both its design and testing.

Debugging

If testing reveals a problem with a script, the next step is debugging. “A problem” usually means that the script is, in some way, not performing to the programmer's expectations. If this is the case, we need to carefully determine exactly what the script is actually doing and why. Finding bugs can sometimes involve a lot of detective work.

A well designed script will try to help. It should be programmed defensively, to detect abnormal conditions and provide useful feedback to the user. Sometimes, however, problems are quite strange and unexpected, and more involved techniques are required.

Finding The Problem Area

In some scripts, particularly long ones, it is sometimes useful to isolate the area of the script that is related to the problem. This won’t always be the actual error, but isolation will often provide insights into the actual cause. One technique that can be used to isolate code is “commenting out” sections of a script. For example, our file deletion fragment could be modified to determine if the removed section was related to an error:

```
        N     T
           N   T
             D

              _          J   N   J_ CY*
              "

U
U       J          !2 %3   .   %J &'
U
```

By placing comment symbols at the beginning of each line in a logical section of a script, we prevent that section from being executed. Testing can then be performed again, to see if the removal of the code has any impact on the behavior of the bug.

Tracing

Bugs are often cases of unexpected logical flow within a script. That is, portions of the script are either never being executed, or are being executed in the wrong order or at the wrong time. To view the actual flow of the program, we use a technique called .

One tracing method involves placing informative messages in a script that display the location of execution. We can add messages to our code fragment:

```
J                J &'
     N     T
        N   T
J         J &'
          D

            _        J   N   J_ CY*
            "


     _           , J   N   J_ CY*
     "

J              J &'
```

We send the messages to standard error to separate them from normal output. We also do not indent the lines containing the messages, so it is easier to find when it's time to remove them.

Now when the script is executed, it's possible to see that the file deletion has been performed:

```
                        6
6 6

         6
```

also provides a method of tracing, implemented by the option and the command with the option. Using our earlier script, we can activate tracing for the entire script by adding the option to the first line:

```
UL
```

```
Z          6

   L"

       L "  T
       _K        A      "<_

       _K           A      "<_
```

When executed, the results look like this:

```
7      L"
7 J J " L " J J
7     JK         A      "<J
K        A      "<
```

With tracing enabled, we see the commands performed with expansions applied. The leading plus signs indicate the display of the trace to distinguish them from lines of regular output. The plus sign is the default character for trace output. It is contained in the :(& (prompt string 4) shell variable. The contents of this variable can be adjusted to make the prompt more useful. Here, we modify the contents of the variable to include the current line number in the script where the trace is performed. Note that single quotes are required to prevent expansion until the prompt is actually used:

```
                              BFA$%3TP?G?S 5 %

# 7      L"
0 7 J J " L " J J
! 7     JK         A      "<J
K        A      "<
```

To perform a trace on a selected portion of a script, rather than the entire script, we can use the command with the option:

```
ZF5  5

Z          6

   L"
```

```
   U +
      L " T
      _K       A     "<_

      _K          A     "<_

  5  U +
```

We use the command with the option to activate tracing and the 7 option to deactivate tracing. This technique can be used to examine multiple portions of a troublesome script.

Examining Values During Execution

It is often useful, along with tracing, to display the content of variables to see the internal workings of a script while it is being executed. Applying additional statements will usually do the trick:

```
ZF5  5

Z         6

   L"

   J    $3     J U 1G<EQ
    Z )
      L " T
      _K       A     "<_

      _K          A     "<_

  7  Z )
```

In this trivial example, we simply display the value of the variable number and mark the added line with a comment to facilitate its later identification and removal. This technique is particularly useful when watching the behavior of loops and arithmetic within scripts.

Summing Up

In this chapter, we looked at just a few of the problems that can crop up during script development. Of course, there are many more. The techniques described here will enable finding most common bugs. Debugging is a fine art that is developed through experience,

both in knowing how to avoid bugs (testing constantly throughout development) and in finding bugs (effective use of tracing).

Further Reading

The Wikipedia has a couple of short articles on syntactic and logical errors:
http://en.wikipedia.org/wiki/Syntax_error
http://en.wikipedia.org/wiki/Logic_error

There are many online resources for the technical aspects of programming:
http://mywiki.wooledge.org/BashPitfalls
http://tldp.org/LDP/abs/html/gotchas.html
http://www.gnu.org/software/bash/manual/html_node/Reserved-Word-Index.html

Eric Raymond's * 0 7 is a great resource for learning the basic concepts found in well-written Unix programs. Many of these ideas apply to shell scripts:
http://www.faqs.org/docs/artu/
http://www.faqs.org/docs/artu/ch01s06.html

For really heavy-duty debugging, there is the Bash Debugger:
http://bashdb.sourceforge.net/

' 5 ; 6

In this chapter, we will continue our look at flow control. In Chapter 28, we constructed some simple menus and built the logic used to act on a user's selection. To do this, we used a series of commands to identify which of the possible choices has been selected. This type of logical construct appears frequently in programs, so much so that many programming languages (including the shell) provide a special flow control mechanism for multiple-choice decisions.

In , multiple-choice compound command is called . It has the following syntax:

P <<<O TT <<<

If we look at the program from Chapter 28, we see the logic used to act on a user's selection:

```
ZF5  5

Z                  ,          6



   _
:   (

"< 9  6  , (,    @
*< 9  6  , 9   (6
/< 9  6  , R   (6   2   B
$< [
_
    6 _'           $ /  C _

  (( 3HGBT^ $  [(*  )3 ))#
        (( 3HGBT^ $$ * ))#
```

```
                JB              J


        (( 3HGBT^ $$   ))#
                JR       2 3RSF+?:OGJ



        (( 3HGBT^ $$   ))#



        (( 3HGBT^ $$   ))#
               (( 34     6   , * ))#
                        JR    F     E    -      4:   E    6J


                        JR    F     E    -      43EFGH6J
                          3RSOG




          JP          ! J &'
```

Using , we can replace this logic with something simpler:

```
ZF5   5

Z                    ,              6



    _
:     (

"< 9  6  , (,     @
*< 9  6  , 9    (6
/< 9  6  , R    (6    2    B
$< [
_
     6 _'             $ /  C _

    3HGBT^
      *6          JB              J

              ##
```

```
6          JR      2 3RSF+?:OGJ

       ##
6
       ##
6        (( 34    6   , * ))#
                  JR   F    E   -    4:  E   6J


                  JR   F    E   -    43EFGH6J
                   3RSOG

       ##
6          JP        !J &'

       ##
```

The command looks at the value of , in our example, the value of the Q':=U variable, and then attempts to match it against one of the specified . When a match is found, the associated with the specified pattern are executed. After a match is found, no further matches are attempted.

Patterns

The patterns used by are the same as those used by pathname expansion. Patterns are terminated with a “)” character. Here are some valid patterns:

! (7 1

Pattern	Description
O	Matches if equals “a”.
6 O	Matches if is a single alphabetic character.
EEEO	Matches if is exactly three characters long.
D< O	Matches if ends with the characters “.txt”.
DO	Matches any value of . It is good practice to include this as the last pattern in a command, to catch any values of that did not match a previous pattern; that is, to catch any possible invalid values.

Here is an example of patterns at work:

```
ZF5   5

    6 _     8    C _

    Q':=U
         6    O                _          6            <_ TT
      3G   $ % O               _   3> G>        8    ,      <_ TT
      EEEO                     _                <_ TT
      D<   O                   _    8          J<   J_ TT
      DO                       _            <_ TT
```

It is also possible to combine multiple patterns using the vertical bar character as a separator. This creates an “or” conditional pattern. This is useful for such things as handling both upper- and lowercase characters. For example:

```
ZF5   5

Z                    ,            6



    _
:    (

:    9  6  , (,     @
<    9  6  , 9    (6
=    9  6  , R    (6    2    B
\    [
_
    6 _'                              (:; <; =    \)            C _

    Q':=U
      ,0\6        _:              <_

              TT
      0:6         _R         RS()K3-'_
               6
              TT
      0<6
              TT
      0=6         M     O   A $   T
                          _R    (6    2    B      M3   2    O_
                            5    5D

                          _R    (6    2    B      M 2('QO_
                             RS-'
```

```
          TT
    DO        _@          ,_ CY*
              "
          TT
```

Here, we modify the program to use letters instead of digits for menu selection. Notice how the new patterns allow for entry of both upper- and lowercase letters.

Performing Multiple Actions

In versions of prior to 4.0, allowed only one action to be performed on a successful match. After a successful match, the command would terminate. Here we see a script that tests a character:

```
ZF5   5

Z    & "

     "  6 _),6          C _

    Q':=U
     66    O       _J Q':=UJ    66      <_ TT
      8    O       _J Q':=UJ    8      <_ TT
      6    O       _J Q':=UJ    6      <_ TT
           O       _J Q':=UJ        <_ TT
      6    O       _J Q':=UJ                <_ TT
     6     O       _J Q':=UJ    6         ,   <_ TT
     6     O       _J Q':=UJ    8    6          <_ TT
           O       _J Q':=UJ                <_ TT
```

Running this script produces this:

```
                          A
),6          C
J J    8     <
```

The script works for the most part, but fails if a character matches more than one of the POSIX characters classes. For example, the character "a" is both lower case and alphabetic, as well as a hexadecimal digit. In prior to version 4.0 there was no way for to match more than one test. Modern versions of , add the “;;&” notation to

terminate each action, so now we can do this:

```
ZF5  5

Z   & *

    "  6 _),6          C _

   Q':=U
    66   O      _J Q':=UJ   66     <_ TTY
     8   O      _J Q':=UJ   8     <_ TTY
     6   O      _J Q':=UJ   6     <_ TTY
         O      _J Q':=UJ       <_ TTY
      6  O      _J Q':=UJ             <_ TTY
    6    O      _J Q':=UJ   6       ,   <_ TTY
     6   O      _J Q':=UJ   8   6         <_ TTY
         O      _J Q':=UJ             <_ TTY
```

When we run this script, we get this:

```
                      A
),6         C
J J    8     <
J J    6     <
J J              <
J J              <
```

The addition of the ";;&" syntax allows case to continue on to the next test rather than simply terminating.

Summing Up

The command is a handy addition to our bag of programming tricks. As we will see in the next chapter, it's the perfect tool for handling certain types of problems.

Further Reading

The 9 : % section on Conditional Constructs describes the command in detail:
http://tiswww.case.edu/php/chet/bash/bashref.html#SEC21

The * 9 # ? provides further examples of applica-

tions:
http://tldp.org/LDP/abs/html/testbranch.html

2 2 $

One feature that has been missing from our programs, so far, is the ability to accept and process command line options and arguments. In this chapter, we will examine the shell features that allow our programs to get access to the contents of the command line.

Accessing The Command Line

The shell provides a set of variables called that contain the individual words on the command line. The variables are named $ through % They can be demonstrated this way:

```
ZF5  5

Z 6   6     6    8       6


^ $ L $
^ " L "
^ * L *
^ / L /
^ & L &
^ # L #
^ + L +
^ 0 L 0
^ ! L !
^ % L %
_
```

A very simple script that displays the values of the variables $ % . When executed with no command line arguments, the result is this:

```


$ L 5   5 5  56   6
```

```
" L
* L
/ L
& L
# L
+ L
0 L
! L
% L
```

Even when no arguments are provided, $ will always contain the first item appearing on the command line, which is the pathname of the program being executed. When arguments are provided, we see these results:

```
$ L 5  5 5  56   6
" L
* L
/ L
& L
# L
+ L
0 L
! L
% L
```

2 6 You can actually access more than nine parameters using parameter expansion. To specify a number greater than nine, surround the number in braces. For example ${10}, ${55}, ${211}, and so on.

Determining The Number of Arguments

The shell also provides a variable, Z , that contains the number of arguments on the command line:

```
ZF5  5

Z 6   6     6    8       6

  –
```

```
?          2 3U
^ $ L  $
^ " L  "
^ * L  *
^ / L  /
^ & L  &
^ # L  #
^ + L  +
^ 0 L  0
^ ! L  !
^ % L  %
_
```

The result:

```
K           &
 $ L 5   5 5  56    6
 " L
 * L
 / L
 & L
 # L
 + L
 0 L
 ! L
 % L
```

– Getting Access To Many Arguments

But what happens when we give the program a large number of arguments such as this:

```
K           !*
 $ L 5   5 5  56    6
 " L       <
 * L
 / L       <
 & L      #$$  /!+      <
 # L      #$$  /!+      <
 + L      #$$  /!+      <   6
 0 L       N   <   < B
```

```
! L 9   6
 % L       <
```

On this example system, the wildcard D expands into 82 arguments. How can we process that many? The shell provides a method, albeit a clumsy one, to do this. The command causes all the parameters to "move down one" each time it is executed. In fact, by using , it is possible to get by with only one parameter (in addition to $, which never changes):

```
ZF5  5

Z 6   6  *   6     6 ,

   L"

8      Z   $  T
        _3          L  "_
        L MM     7 "OO
```

Each time is executed, the value of * is moved to " , the value of / is moved to * and so on. The value of Z is also reduced by one.

In the 6 6 * program, we create a loop that evaluates the number of arguments remaining and continues as long as there is at least one. We display the current argument, increment the variable with each iteration of the loop to provide a running count of the number of arguments processed and, finally, execute a to load " with the next argument. Here is the program at work:

```


3      " L
3      * L
3      / L
3      & L
```

Simple Applications

Even without , it's possible to write useful applications using positional parameters. By way of example, here is a simple file information program:

```
ZF5  5

Z   N      6            6

:QS?K3-'L M       $O

      "  T
         _^     ),6  _
        "
         _^     (     _
        "

        _ :QS?K3-'       :QS?K3-'     _ CY*
        "
```

This program displays the file type (determined by the command) and the file status (from the command) of a specified file. One interesting feature of this program is the :QS?K3-' variable. It is given the value that results from the $ command. The command removes the leading portion of a pathname, leaving only the base name of a file. In our example, removes the leading portion of the pathname contained in the $ parameter, the full pathname of our example program. This value is useful when constructing messages such as the usage message at the end of the program. By coding it this way, the script can be renamed and the message automatically adjusts to contain the name of the program.

Using Positional Parameters With Shell Functions

Just as positional parameters are used to pass arguments to shell scripts, they can also be used to pass arguments to shell functions. To demonstrate, we will convert the N script into a shell function:

```
 N    MO \

   Z    N              6  ,

        "  T
            _^     ),6  _
           "
            _^     (     _
           "

           _ 2K K3-'         2K K3-'     _ CY*
           "
```

```
]
```

Now, if a script that incorporates the N shell function calls the function with a filename argument, the argument will be passed to the function.

With this capability, we can write many useful shell functions that can not only be used in scripts, but also within our < files.

Notice that the :QS?K3-' variable was changed to the shell variable 2K K3-' . The shell automatically updates this variable to keep track of the currently executed shell function. Note that $ always contains the full pathname of the first item on the command line (i.e., the name of the program) and does not contain the name of the shell function as we might expect.

Handling Positional Parameters ! &

It is sometimes useful to manage all the positional parameters as a group. For example, we might want to write a “wrapper” around another program. This means that we create a script or shell function that simplifies the invocation of another program. The wrapper, in this case, supplies a list of arcane command line options and then passes a list of arguments to the lower-level program.

The shell provides two special parameters for this purpose. They both expand into the complete list of positional parameters, but differ in rather subtle ways. They are:

! (" Q * # 7

Parameter	Description
D	Expands into the list of positional parameters, starting with 1. When surrounded by double quotes, it expands into a double quoted string containing all of the positional parameters, each separated by the first character of the IFS shell variable (by default a space character).
	Expands into the list of positional parameters, starting with 1. When surrounded by double quotes, it expands each positional parameter into a separate word surrounded by double quotes.

Here is a script that shows these special paramaters in action:

```
ZF5  5

Z 6   6   /   6          D

6  N6    MO \
       _^ " L  "_
       _^ * L  *_
       _^ / L  /_
       _^ & L  &_
]

6  N6    MO \
        _^ _ J D  JT   6   N6     D
        _^ _ J_ D_  JT 6   N6     _ D_
        _^ _ J   JT   6   N6
        _^ _ J_  _  JT 6   N6     _  _
]

6  N6    _8  _ _8   8   6   _
```

In this rather convoluted program, we create two arguments: “word” and “words with spaces”, and pass them to the 6 N6 function. That function, in turn, passes them on to the 6 N6 function, using each of the four methods available with the special parameters D and . When executed, the script reveals the differences:

```
 D
" L 8
* L 8
/ L 8
& L  6

_ D_
" L 8   8   8   6
* L
/ L
& L


" L 8
* L 8
/ L 8
& L  6


_ _
" L 8
```

```
*L8   8   6
/L
&L
```

With our arguments, both D and produce a four word result:

8 8 8 6

_D_ produces a one word result:

_8 8 8 6 _

_ _ produces a two word result:

_8 _ _8 8 6 _

which matches our actual intent. The lesson to take from this is that even though the shell provides four different ways of getting the list of positional parameters, _ _ is by far the most useful for most situations, because it preserves the integrity of each positional parameter.

A More Complete Application

After a long hiatus, we are going to resume work on our , N N6 program. Our next addition will add several command line options to the program as follows:

; (" We will add an option to specify a name for a file to contain the program's output. It will be specified as either or .

) This option will prompt the user for an output filename and will determine if the specified file already exists. If it does, the user will be prompted before the existing file is overwritten. This option will be specified by either or .

(Either or 6 may be specified to cause the program to output an informative usage message.

Here is the code needed to implement the command line processing:

```
   MO \
      _ :QS?K3-'      :QS?K3-'      P  _

]

Z6            6

      L
```

```
     L

8        "  T
         "
             P     O
                                        L "
                                  TT
             P          O                  L"
                                  TT
             P    6O

                                  TT
           DO                        CY*
                                    "
                                  TT
```

First, we add a shell function called to display a message when the help option is invoked or an unknown option is attempted.

Next, we begin the processing loop. This loop continues while the positional parameter " is not empty. At the end of the loop, we have a command to advance the positional parameters to ensure that the loop will eventually terminate.

Within the loop, we have a statement that examines the current positional parameter to see if it matches any of the supported choices. If a supported parameter is found, it is acted upon. If an unknown choice is founc, the usage message is displayed and the script terminates with an error.

The parameter is handled in an interesting way. When detected, it causes an additional to occur, which advances the positional parameter " to the filename argument supplied to the option.

We next add the code to implement the interactive mode:

```
Z

              T
     8      T
              6 _'          6       _
                     T
                   6 _J       J     < S   8    E  ,5 5A  C _
                   Q':=U
                      UP,O
```

```
                    TT
                [PAO    _:          <_

                    TT
                DO
                    TT

            B       T
```

If the variable is not empty, an endless loop is started, which contains the filename prompt and subsequent existing file-handling code. If the desired output file already exists, the user is prompted to overwrite, choose another filename, or quit the program. If the user chooses to overwrite an existing file, a is executed to terminate the loop. Notice how the statement only detects if the user chooses to overwrite or quit. Any other choice causes the loop to continue and prompts the user again.

In order to implement the output filename feature, we must first convert the existing page-writing code into a shell function, for reasons that will become clear in a moment:

```
8   N   N6   MO \
        XX  N'S N
     XR)-=C
            XR'39C
                    X)@)='C )@)='X5)@)='C
            X5R'39C
            XGS9UC
                    XR"C )@)='X5R"C
                    X:C )@-'()3-:X5:C
                     M  6   N 6   O
                     M  6   N   N 6   O
                     M  6   N   N 6   O
            X5GS9UC
     X5R)-=C
     N'S N

]

Z   6      6

              T
```

```
                YY              T
          8   N   N6   C

             _ :QS?K3-'       8        J       J_ CY*
             "


     8   N   N6
```

The code that handles the logic of the option appears at the end of the listing shown above. In it, we test for the existence of a filename and, if one is found, a test is performed to see if the file is indeed writable. To do this, a is performed, followed by a test to determine if the resulting file is a regular file. These two tests take care of situations where an invalid pathname is input (will fail), and, if the file already exists, that it's a regular file.

As we can see, the 8 N N6 function is called to perform the actual generation of the page. Its output is either directed to standard output (if the variable is empty) or redirected to the specified file.

Summing Up

With the addition of positional parameters, we can now write fairly functional scripts. For simple, repetitive tasks, positional parameters make it possible to write very useful shell functions that can be placed in a user's < file.

Our , N N6 program has grown in complexity and sophistication. Here is a complete listing, with the most recent changes highlighted:

```
ZF5   5

Z  , N    N6    6          6     ,            6

BHSQ?:OG$34        3*6
)@)='L_(,     @          Q 6        RS()K3-'_
 2QQ'K)N)@-'L M     7_4  4  4H_O
)@-'()3-:L_?          2QQ'K)N)@-'>  ,  2('Q_

  6   N 6     MO \
         XX  N'S N
              XR*C(,     26    X5R*C
              X:Q'C M 6    OX5:Q'C
              N'S N
```

```
]

 6   N    N 6    MO \
        XX  N'S N
            XR*C9    (6    2    B     X5R*C
            X:Q'C M     OX5:Q'C
            N'S N

]

 6   N    N 6    MO \
          M     O   A $   T
              XX  N'S N
                  XR*CR    (6    2    B      M3   2    OX5R*C
                  X:Q'C M      5    5DOX5:Q'C
                  N'S N

              XX  N'S N
                  XR*CR    (6    2    B      M 2('QOX5R*C
                  X:Q'C M       RS-'OX5:Q'C
                  N'S N


]

    46 9
        J3BHSQ?:OG2      2 3BHSQ?:OG (       0   )J

>

   .   .    46 9
        //  .GS7.
      /R+OT&
              /RG:1&
                      /+P+TG&3+P+TG/ +P+TG&
              / RG:1&
              /<S1^&
                      /R &3+P+TG/ R &
                      /B&3+POGF+:OB/ B&
                      34      .      6
                      34      .    .     6
                      34      .    .     6
              / <S1^&
      / R+OT&
      .GS7.

>

Z 6                 6
```

```
        $
      $

    ((    3  ))#
          3
                0      6
                                                $3
                                          ##
                0           6                    $
                                          ##
                0      6

                                          ##
              6                              &'

                                          ##




Z

  ((    3          ))#
              #
                   JG                  2 J
                ((    3        ))#
                          J%3       %        S        _ (!   ,) & J
                        3HGBT^
                            ^0!6
                                  ##
                            \0,6        JB                J

                                  ##
                             6
                                  ##

                 ((  - 3        ))#





Z   6       6

  ((    3        ))#
              3        " ((    3        ))#
                  .    .     & 3
```

```
J3BHSQ?:OG2 =          %3      %J &'

.  .
```

We're not done yet. There are still more things we can do and improvements we can make.

Further Reading

The 9 , / has a good article on positional parameters: http://wiki.bash-hackers.org/scripting/posparams

The 9 : % has an article on the special parameters, including D and : http://www.gnu.org/software/bash/manual/bashref.html#Special-Parameters

In addition to the techniques discussed in this chapter, includes a builtin command called 6 , which can also be used for process command line arguments. It is described in the SHELL BUILTIN COMMANDS section of the man page and at the 9 , / : http://wiki.bash-hackers.org/howto/getopts_tutorial

' 5 ; < "

In this final chapter on flow control, we will look at another of the shell's looping constructs. The differs from the while and until loops in that it provides a means of processing sequences during a loop. This turns out to be very useful when programming. Accordingly, the for loop is a very popular construct in scripting.

A for loop is implemented, naturally enough, with the compound command. In modern versions of , is available in two forms.

: Traditional Shell Form

The original command's syntax is:

T

Where ! is the name of a variable that will increment during the execution of the loop, is an optional list of items that will be sequentially assigned to ! , and are the commands that are to be executed on each iteration of the loop.

The command is useful on the command line. We can easily demonstrate how it works:

```
: < = 1#  3 #
3
G

9
```

In this example, is given a list of four words: "A", "B", "C", and "D". With a list of four words, the loop is executed four times. Each time the loop is executed, a word is assigned to the variable . Inside the loop, we have an command that displays the value of to show the assignment. As with the 8 and loops, the keyword closes the loop.

The really powerful feature of is the number of interesting ways we can create the list of words. For example, through brace expansion:

```
                          9:  1>#       3 #
3
G

9
```

or pathname expansion:

```
                                  #       3 #
     ,   <
       <
     ,     <
     ,       <
       <
   <
         <
          <
```

or command substitution:

```
ZF5  5

Z      8

8        "  T
           "  T
             N8  L
             N  L$
                 M      "OT
                   L M        P 8   O
                   MM    C    N   OOT
                         N  L
                         N8  L


              _ " J   N8  J M   N          O_
```

In this example, we look for the longest string found within a file. When given one or more filenames on the command line, this program uses the program (which is included in the GNU binutils package) to generate a list of readable text “words” in each file. The loop processes each word in turn and determines if the current word is the longest found so far. When the loop concludes, the longest word is displayed.

If the optional portion of the command is omitted, defaults to processing the positional parameters. We will modify our 8 script to use this method:

```
ZF5  5

Z     8  *

  T
          T
          N8  L
          N  L$
              M       OT
                L M       P 8   O
                MM   C   N   OOT
                     N  L
                     N8  L


           _  J   N8  J M   N          O_
```

As we can see, we have changed the outermost loop to use in place of 8 . By omitting the list of words in the command, the positional parameters are used instead. Inside the loop, previous instances of the variable have been changed to the variable . The use of has also been eliminated.

Why i?

You may have noticed that the variable was chosen for each of the loop examples above. Why? No specific reason actually, besides tradition. The variable used with can be any valid variable, but is the most common, followed by and .

The basis of this tradition comes from the Fortran programming language. In Fortran, undeclared variables starting with the letters I, J, K, L, and M are automatically typed as integers, while variables beginning with any other letter are typed as real (numbers with decimal fractions). This behavior led programmers to use the variables I, J, and K for loop variables, since it was less work to use them when a temporary variable (as loop variables often are) was needed.

It also led to the following Fortran-based witticism:

“GOD is real, unless declared integer.”

: C Language Form

Recent versions of have added a second form of command syntax, one that resembles the form found in the C programming language. Many other languages support this form, as well:

```
MM                T                T                OOT
```

where , " , and (are arithmetic expressions and are the commands to be performed during each iteration of the loop.

In terms of behavior, this form is equivalent to the following construct:

```
MM                OO
8   MM                OOT

        MM                OO
```

is used to initialize conditions for the loop, " is used to determine when the loop is finished, and (is carried out at the end of each iteration of the loop.

Here is a typical application:

```
ZF5  5

Z   6  N              ,

   MM  L$T  X#T  L 7" OOT
```

When executed, it produces the following output:

```
                        .
$
"
*
/
&
```

In this example, initializes the variable with the value of zero, " allows the loop to continue as long as the value of remains less than 5, and (increments the value of by one each time the loop repeats.

The C language form of is useful anytime a numeric sequence is needed. We will see several applications for this in the next two chapters.

Summing Up

With our knowledge of the command, we will now apply the final improvements to our , N N6 script. Currently, the 6 N N 6 function looks like this:

```
 6  N   N 6   MO \
       M   O  A $  T
           XX  N'S N
               XR*CR   (6   2   B     M3  2   OX5R*C
               X:Q'C M     5   5DOX5:Q'C
               N'S N

           XX  N'S N
               XR*CR   (6   2   B     M 2('QOX5R*C
               X:Q'C M      RS-'OX5:Q'C
               N'S N


]
```

Next, we will rewrite it to provide more detail for each user's home directory, and include the total number of files and subdirectories in each:

```
 6  N   N 6   MO \
```

```
            L_4! 4"$ 4"$ ^ _
           N       N       N       N  B     N

         M    O   A $   T
              N   L5    5D
               N   L_3   2    _

              N   L RS-'
               N   L 2('Q


        _XR*CR    (6    2    B     M     N    OX5R*C_

             N   T

               N    L M        ,6    P 8    O
               N    L M        ,6    P 8    O
               N  B L M        P       "O

               _XR/C  X5R/C_
               _X:Q'C_
            6    _     _ _9   _ _    _ _( B _
            6    _     _ _    _ _    _ _   _
            6    _     _      N       N        N  B
               _X5:Q'C_


]
```

This rewrite applies much of what we have learned so far. We still test for the superuser, but instead of performing the complete set of actions as part of the , we set some variables used later in a for loop. We have added several local variables to the function and made use of 6 to format some of the output.

Further Reading

The * 9 # ? has a chapter on loops, with a variety of examples using :
http://tldp.org/LDP/abs/html/loops1.html

The 9 : % describes the looping compound commands, including :
http://www.gnu.org/software/bash/manual/bashref.html#Looping-Constructs

% ($

Computer programs are all about working with data. In past chapters, we have focused on processing data at the file level. However, many programming problems need to be solved using smaller units of data such as strings and numbers.

In this chapter, we will look at several shell features that are used to manipulate strings and numbers. The shell provides a variety of parameter expansions that perform string operations. In addition to arithmetic expansion (which we touched upon in Chapter 7), there is a well-known command line program called , which performs higher level math.

Parameter Expansion

Though parameter expansion came up in Chapter 7, we did not cover it in detail because most parameter expansions are used in scripts rather than on the command line. We have already worked with some forms of parameter expansion; for example, shell variables. The shell provides many more.

Basic Parameters

The simplest form of parameter expansion is reflected in the ordinary use of variables. For example:

when expanded, becomes whatever the variable contains. Simple parameters may also be surrounded by braces:

```
\]
```

This has no effect on the expansion, but is required if the variable is adjacent to other text, which may confuse the shell. In this example, we attempt to create a filename by appending the string "_file" to the contents of the variable .

```
$J  J
  J3 .  J
```

```

```

If we perform this sequence of commands, the result will be nothing, because the shell will try to expand a variable named N rather than . This problem can be solved by adding braces around the “real” variable name:

```
                    J39 >.  J
N
```

We have also seen that positional parameters greater than 9 can be accessed by surrounding the number in braces. For example, to access the eleventh positional parameter, we can do this:

\""]

Expansions To Manage Empty Variables

Several parameter expansions are intended to deal with nonexistent and empty variables. These expansions are handy for handling missing positional parameters and assigning default values to parameters.

\ ]

If is unset (i.e., does not exist) or is empty, this expansion results in the value of . If is not empty, the expansion results in the value of .

```
                    $
                     39  2 J          J>

                     3

                    $
                     39  2 J          J>

                     3
```

\ L]

If is unset or empty, this expansion results in the value of . In addition, the value of is assigned to . If is not empty, the expansion results in the value of .

```
                    $
                     39  2$J             J>

                     3

                    $
                     39  2$J             J>

                     3
```

2 : Positional and other special parameters cannot be assigned this way.

\ E]

If is unset or empty, this expansion causes the script to exit with an error, and the contents of are sent to standard error. If is not empty, the expansion results in the value of .

```
                    $
                     39  2_J          !J>
      6        6 ,
                     3_
"
                    $
                     39  2_J          !J>

                     3_
$
```

\ 7]

If is unset or empty, the expansion results in nothing. If is not empty, the value of is substituted for ; however, the value of is not changed.

```
                    $
                     39  25J             J>

                    $
                     39  25J             J>
```

Expansions That Return Variable Names

The shell has the ability to return the names of variables. This is used in some rather exotic situations.

\F D]
\F]

This expansion returns the names of existing variables with names beginning with . According to the documentation, both forms of the expansion perform identically. Here, we list all the variables in the environment with names that begin with BASH:

```
39L<:FR >
G3(R G3(RN3Q?  G3(RN3Q?; G3(RN S--3K9 G3(RN S-:=')@SK
G3(RN S-:=')@SKN9@Q G3(RN=@K'KS G3(RN(S2Q ' G3(RN(2G(R'==
G3(RN;'Q(@K S G3(RN;'Q(@SK
```

String Operations

There is a large set of expansions that can be used to operate on strings. Many of these expansions are particularly well suited for operations on pathnames.

\Z]

expands into the length of the string contained by . Normally, is a string; however, if is either or D, then the expansion results in the number of positional parameters.

```
$J+ J
J%3 % 39U > J
J) <J *$ <
```

\ ]
\ ]

These expansions are used to extract a portion of the string contained in . The extraction begins at characters from the beginning of the string and continues until the end of the string, unless the is specified.

```
                    $J+          J
                     39  2">
       <
                     39  2"2N>
```

If the value of is negative, it is taken to mean it starts from the end of the string rather than the beginning. Note that negative values must be preceded by a space to prevent confusion with the \ ] expansion. , if present, must not be less than zero.

If is , the result of the expansion is positional parameters, starting at .

```
                    $J+          J
                     39  2 ">
  <
                     39  2 "2 >
```

\ Z]
\ ZZ]

These expansions remove a leading portion of the string contained in defined by . is a wildcard pattern like those used in pathname expansion. The difference in the two forms is that the Z form removes the shortest match, while the ZZ form removes the longest match.

```
                    $     -
                     39  U >
  <B 6
                     39  UU >
B 6
```

\ 4]
\ 44]

These expansions are the same as the Z and ZZ expansions above, except they remove text from the end of the string contained in rather than from the beginning.

```
                    $     -
```

```
                    39  8 >
  <
                    39  88 >
```

```
\          5          5          ]
\          55          5          ]
\          5Z          5          ]
\          54          5          ]
```

This expansion performs a search-and-replace upon the contents of . If text is found matching wildcard , it is replaced with the contents of . In the normal form, only the first occurrence of is replaced. In the 55 form, all occurrences are replaced. The 5Z form requires that the match occur at the beginning of the string, and the 54 form requires the match to occur at the end of the string. In every form, / may be omitted, causing the text matched by to be deleted.

```
                    $YBQ YBQ
                     39   YBQ   >
6 < :?
                     39    YBQ   >
6 < 6
                     39   UYBQ   >
6 < :?
                     39   8YBQ   >
:?< 6
```

Parameter expansion is a good thing to know. The string manipulation expansions can be used as substitutes for other common commands such as and . Expansions can improve the efficiency of scripts by eliminating the use of external programs. As an example, we will modify the 8 program discussed in the previous chapter to use the parameter expansion \Z] in place of the command substitution M P 8 O and its resulting subshell, like so:

```
ZF5  5

Z     8  /

  T
          T
          N8  L
          N  L$
```

```
                M        OT
                  $39U >
                  MM    C    N    OOT
                        N   L
                        N8   L


             _   J    N8   J M    N           O_
```

Next, we will compare the efficiency of the two versions by using the command:

```
          <    J       6                  J M/!
      O

           $ /<+"!
           $ "<#&&
,     $ "<0+!

          <    J       6                  J M/!
      O

           $ $<$+$
           $ $<$#+
,     $ $<$$!
```

The original version of the script takes 3.618 seconds to scan the text file, while the new version, using parameter expansion, takes only 0.06 seconds—a very significant improvement.

Case Conversion

Recent versions of have support for upper/lowercase conversion of strings. has four parameter expansions and two options to the command to support it.

So what is case conversion good for? Aside from the obvious aesthetic value, it has an important role in programming. Let's consider the case of a database look-up. Imagine that a user has entered a string into a data input field that we want to look up in a database. It's possible the user will enter the value in all uppercase letters or lowercase letters or a combination of both. We certainly don't want to populate our database with every possible permutation of upper and lower case spellings. What to do?

A common approach to this problem is to 8 the user's input. That is, convert it

into a standardized form before we attempt the database look-up. We can do this by converting all of the characters in the user's input to either lower or uppercase and ensure that the database entries are normalized the same way.

The command can be used to normalize strings to either upper or lowercase. Using , we can force a variable to always contain the desired format no matter what is assigned to it:

```
ZF5  5

Z


       66
        8


    "  T
     66 L_"_
      8 L_"_
         66
          8
```

In the above script, we use to create two variables, 66 and 8 . We assign the value of the first command line argument (positional parameter 1) to each of the variables and then display them on the screen:

```
                          <
3G
```

As we can see, the command line argument ("aBc") has been normalized.

In addition to , there are four parameter expansions that perform upper/lowercase conversion:

! (- 7 1

Format		Result	
\	>>]	Expand the value of	into all lowercase.
\	>]	Expand the value of character to lowercase.	changing only the first
\	``]	Expand the value of	into all uppercase letters.

\	`]	Expand the value of changing only the first character to uppercase (capitalization).

Here is a script that demonstrates these expansions:

```
ZF5  5

Z  6                    6       6

    "  T
        \">>]
        \">]
        \"``]
        \"`]
```

Here is the script in action:

```
                        <


 G
3G
3G
```

Again, we process the first command line argument and output the four variations supported by the parameter expansions. While this script uses the first positional parameter, may be any string, variable, or string expression.

Arithmetic Evaluation And Expansion

We looked at arithmetic expansion in Chapter 7. It is used to perform various arithmetic operations on integers. Its basic form is:

MM OO

where is a valid arithmetic expression.

This is related to the compound command MM OOused for arithmetic evaluation (truth tests) we encountered in Chapter 27.

In previous chapters, we saw some of the common types of expressions and operators. Here, we will look at a more complete list.

Number Bases

Back in Chapter 9, we got a look at octal (base 8) and hexadecimal (base 16) numbers. In arithmetic expressions, the shell supports integer constants in any base.

! (- " # . > ! 9

Notation	Description
	By default, numbers without any notation are treated as decimal (base 10) integers.
$	In arithmetic expressions, numbers with a leading zero are considered octal.
$	Hexadecimal notation
Z	! is in !

Some examples:

```
                    344*  66
*##
                    344 U     66
*##
```

In the examples above, we print the value of the hexadecimal number (the largest two-digit number) and the largest eight-digit binary (base 2) number.

Unary Operators

There are two unary operators, the 7 and , which are used to indicate if a number is positive or negative, respectively. For example, # .

Simple Arithmetic

The ordinary arithmetic operators are listed in the table below:

! (- (*)

Operator	Description
7	Addition
	Subtraction

D	Multiplication
5	Integer division
DD	Exponentiation
4	Modulo (remainder)

Most of these are self-explanatory, but integer division and modulo require further discussion.

Since the shell’s arithmetic only operates on integers, the results of division are always whole numbers:

```
                    344 "    66
*
```

This makes the determination of a remainder in a division operation more important:

```
                    344 " 8   66
"
```

By using the division and modulo operators, we can determine that 5 divided by 2 results in 2, with a remainder of 1.

Calculating the remainder is useful in loops. It allows an operation to be performed at specified intervals during the loop's execution. In the example below, we display a line of numbers, highlighting each multiple of 5:

```
ZF5   5

Z                   6

  MM  L $T   XL *$T   L   7 "OOT
        L MM  4 #OO
      MM        LL $ OOT
          6     _X4 C _

          6     _4  _


6     _^ _
```

When executed, the results look like this:

```
X$C " * / & X#C + 0 ! % X"$C "" "* "/ "& X"#C "+ "0 "! "% X*$C
```

Assignment

Although its uses may not be immediately apparent, arithmetic expressions may perform assignment. We have performed assignment many times, though in a different context. Each time we give a variable a value, we are performing assignment. We can also do it within arithmetic expressions:

```
                    $
                     3

                    44    $ " 66#        JP        J#
@       <
                     3
#
```

In the example above, we first assign an empty value to the variable and verify that it is indeed empty. Next, we perform an with the compound command MM L # OO This process does two interesting things: 1) it assigns the value of 5 to the variable , and 2) it evaluates to true because was assigned a nonzero value.

2 6 It is important to remember the exact meaning of the L in the expression above. A single L performs assignment. L # says “make equal to 5,” while LL evaluates equivalence. LL # says “does equal 5?” This is a common feature in many programming languages. In the shell, this can be a little confusing because the command accepts a single L for string equivalence. Yet another reason to use the more modern and MM OOcompound commands in place of .

In addition to the L> the shell also provides notations that perform some very useful assignments:

! (- - *)

Notation	Description

	L	Simple assignment. Assigns to .
	7L	Addition. Equivalent to = + '
	L	Subtraction. Equivalent to = – '
	DL	Multiplication. Equivalent to = * '
	5L	Integer division. Equivalent to = / '
	4L	Modulo. Equivalent to = % '
	77	Variable post-increment. Equivalent to = + 1 (however, see discussion below).
	""	Variable post-decrement. Equivalent to = − 1.
77		Variable pre-increment. Equivalent to = + 1.
		Variable pre-decrement. Equivalent to = − 1.

These assignment operators provide a convenient shorthand for many common arithmetic tasks. Of special interest are the increment (77) and decrement (ee) operators, which increase or decrease the value of their parameters by one. This style of notation is taken from the C programming language and has been incorporated into a number of other programming languages, including .

The operators may appear either at the front of a parameter or at the end. While they both either increment or decrement the parameter by one, the two placements have a subtle difference. If placed at the front of the parameter, the parameter is incremented (or decremented) before the parameter is returned. If placed after, the operation is performed
the parameter is returned. This is rather strange, but it is the intended behavior. Here is a demonstration:

```
                    $
                     344  5566
"
                     3
```

```
*
```

If we assign the value of one to the variable and then increment it with the 77 operator placed after the parameter name, is returned with the value of one. However, if we look at the value of the variable a second time, we see the incremented value. If we place the 77 operator in front of the parameter, we get the more expected behavior:

```
                    $
                     34455  66
*
                     3
*
```

For most shell applications, prefixing the operator will be the most useful.

The ++ and -- operators are often used in conjunction with loops. We will make some improvements to our modulo script to tighten it up a bit:

```
ZF5  5

Z    *              6

  MM  L $T   XL *$T          55   OOT
      MM  4  8 "6     LL $ OOT
          6     _X4 C _

          6     _4  _


6    _^ _
```

Bit Operations

One class of operators manipulates numbers in an unusual way. These operators work at the bit level. They are used for certain kinds of low level tasks, often involving setting or reading bit-flags.

! (- 2 9)

Operator	Description
	Bitwise negation. Negate all the bits in a number.

XX	Left bitwise shift. Shift all the bits in a number to the left.
CC	Right bitwise shift. Shift all the bits in a number to the right.
Y	Bitwise AND. Perform an AND operation on all the bits in two numbers.
P	Bitwise OR. Perform an OR operation on all the bits in two numbers.
`	Bitwise XOR. Perform an exclusive OR operation on all the bits in two numbers.

Note that there are also corresponding assignment operators (for example, <<=) for all but bitwise negation.

Here we will demonstrate producing a list of powers of 2, using the left bitwise shift operator:

```
                        44 $*# /M#55 66#        344 // 66#
"
*
&
!
"+
/*
+&
"*!
```

Logic

As we discovered in Chapter 27, the MM OOcompound command supports a variety of comparison operators. There are a few more that can be used to evaluate logic. Here is the complete list:

```
! (-    3          )
```

Operator	Description
XL	Less than or equal to
CL	Greater than or equal to
X	Less than
C	Greater than

LL	Equal to
FL	Not equal to
YY	Logical AND
PP	Logical OR
E	Comparison (ternary) operator. If expression evaluates to be non-zero (arithmetic true) then " , else (.

When used for logical operations, expressions follow the rules of arithmetic logic; that is, expressions that evaluate as zero are considered false, while non-zero expressions are considered true. The MM OO compound command maps the results into the shell's normal exit codes:

```
44 66#      J  J#      J  J#

44*66#      J  J#      J  J#
```

The strangest of the logical operators is the . This operator (which is modeled after the one in the C programming language) performs a standalone logical test. It can be used as a kind of if/then/else statement. It acts on three arithmetic expressions (strings won't work), and if the first expression is true (or non-zero) the second expression is performed. Otherwise, the third expression is performed. We can try this on the command line:

```
$*
44 / _55 2  66
   3
"
44 / _55 2  66
   3
$
```

Here we see a ternary operator in action. This example implements a toggle. Each time the operator is performed, the value of the variable switches from zero to one or vice versa.

Please note that performing assignment within the expressions is not straightforward.

When attempted, will declare an error:

```
                    $*
                    44 / _ 5$ 2  $ 66
MM   X"E 7L"   L"     6                    M
  _ L"_O
```

This problem can be mitigated by surrounding the assignment expression with parentheses:

```
                    44 / _4 5$ 624  $ 666
```

Next, we see a more complete example of using arithmetic operators in a script that produces a simple table of numbers:

```
ZF5   5

Z       6     6                  6

     L$
 L$
6     _ ^  DD*^  DD/^ _
6     _L^ LLLL^ LLLL^ _

    MM       OOT
      L MM DD*OO
      L MM DD/OO
     6     _4 ^ 4 ^ 4 ^ _
     MM X"$E77  M      L"OOO
```

In this script, we implement an until loop based on the value of the variable. Initially, the variable is set to zero (arithmetic false) and we continue the loop until it becomes non-zero. Within the loop, we calculate the square and cube of the counter variable . At the end of the loop, the value of the counter variable is evaluated. If it is less than 10 (the maximum number of iterations), it is incremented by one, else the variable is given the value of one, making arithmetically true, thereby terminating the loop. Running the script gives this result:

```
      DD*   DD/
L     LLLL  LLLL
$     $     $
"     "     "
*     &     !
/     %     *0
&     "+    +&
#     *#    "*#
+     /+    *"+
0     &%    /&/
!     +&    #"*
%     !"    0*%
"$    "$$   "$$$
```

– An Arbitrary Precision Calculator Language

We have seen how the shell can handle many types of integer arithmetic, but what if we need to perform higher math or even just use floating point numbers? The answer is, we can't. At least not directly with the shell. To do this, we need to use an external program. There are several approaches we can take. Embedding Perl or AWK programs is one possible solution, but unfortunately, outside the scope of this book.

Another approach is to use a specialized calculator program. One such program found on many Linux systems is called .

The program reads a file written in its own C-like language and executes it. A script may be a separate file or it may be read from standard input. The language supports quite a few features including variables, loops, and programmer-defined functions. We won't cover entirely here, just enough to get a taste. is well documented by its man page.

Let's start with a simple example. We'll write a script to add 2 plus 2:

```
5D 3   ,   6       6  D5

* 7 *
```

The first line of the script is a comment. uses the same syntax for comments as the C programming language. Comments, which may span multiple lines, begin with 5D and end with D5.

Using

If we save the script above as < , we can run it this way:

```
  "<$+<%&
 6,    "%%" "%%&> "%%0> "%%!> *$$$> *$$&> *$$+    (  8
      > @  <
)           8   8   3G(S=2)'=U KS .3QQ3K)U<
          ,6  I8    ,J<
&
```

If we look carefully, we can see the result at the very bottom, after the copyright message. This message can be suppressed with the A (quiet) option.

can also be used interactively:

```
                          ,
 5
&
,
```

When using interactively, we simply type the calculations we wish to perform, and the results are immediately displayed. The command A ends the interactive session.

It is also possible to pass a script to via standard input:

```
                         /
&
```

The ability to take standard input means that we can use here documents, here strings, and pipes to pass scripts. This is a here string example:

```
                         /// J 5          J
&
```

An Example Script

As a real-world example, we will construct a script that performs a common calculation, monthly loan payments. In the script below, we use a here document to pass a script to
:

```
ZF5   5

Z             6                  ,     6 ,

:QS?K3-'L \$ZZD5] Z 2   6          6

    MO \
        XX  'S
     2      :QS?K3-' :Q@K @:3= @K)'Q'() -SK)R(

     .

     :Q@K @:3=                      <
     @K)'Q'()       3:Q             M04 L $<$0O<
     -SK)R(                      J     <

     'S
]

  MM Z FL /OOT

         "


6    6  L "
     L *
    L /

  XX  'S
         L "$
      L          5 "*
     6 L  6     6
      L
      L 6 D MM  D MM" 7  O `  OO 5 MMM" 7  O `  O   "OO
     6      > _^ _
'S
```

When executed, the results look like this:

```
                                "*** * *CC"  M*
```

```
"*0$<0***&%$$$$
```

This example calculates the monthly payment for a $135,000 loan at 7.75% APR for 180 months (15 years). Notice the precision of the answer. This is determined by the value given to the special variable in the script. A full description of the scripting language is provided by the man page. While its mathematical notation is slightly different from that of the shell (more closely resembles C), most of it will be quite familiar, based on what we have learned so far.

Summing Up

In this chapter, we have learned about many of the little things that can be used to get the "real work" done in scripts. As our experience with scripting grows, the ability to effectively manipulate strings and numbers will prove extremely valuable. Our script demonstrates that even simple scripts can be created to do some really useful things.

Extra Credit

While the basic functionality of the script is in place, the script is far from complete. For extra credit, try improving the script with the following features:

Full verification of the command line arguments

A command line option to implement an "interactive" mode that will prompt the user to input the principal, interest rate, and term of the loan.

A better format for the output.

Further Reading

The 9 , / has a good discussion of parameter expansion:
http://wiki.bash-hackers.org/syntax/pe

The 9 : % covers this, too:
http://www.gnu.org/software/bash/manual/bashref.html#Shell-Parameter-Expansion

The / has a good article describing bit operations:
http://en.wikipedia.org/wiki/Bit_operation

and an article on ternary operations:
http://en.wikipedia.org/wiki/Ternary_operation

as well as a description of the formula for calculating loan payments used in our script:
http://en.wikipedia.org/wiki/Amortization_calculator

* (

In the previous chapter, we looked at how the shell can manipulate strings and numbers. The data types we have looked at so far are known in computer science circles as ! ; that is, variables that contain a single value.

In this chapter, we will look at another kind of data structure called an , which holds multiple values. Arrays are a feature of virtually every programming language. The shell supports them, too, though in a rather limited fashion. Even so, they can be very useful for solving some types of programming problems.

What Are Arrays?

Arrays are variables that hold more than one value at a time. Arrays are organized like a table. Let's consider a spreadsheet as an example. A spreadsheet acts like a . It has both rows and columns, and an individual cell in the spreadsheet can be located according to its row and column address. An array behaves the same way. An array has cells, which are called , and each element contains data. An individual array element is accessed using an address called an or ! .

Most programming languages support . A spreadsheet is an example of a multidimensional array with two dimensions, width and height. Many languages support arrays with an arbitrary number of dimensions, though two- and three-dimensional arrays are probably the most commonly used.

Arrays in are limited to a single dimension. We can think of them as a spreadsheet with a single column. Even with this limitation, there are many applications for them. Array support first appeared in version 2. The original Unix shell program, , did not support arrays at all.

Creating An Array

Array variables are named just like other variables, and are created automatically when they are accessed. Here is an example:

```
                    ( )$
                      39 ( )>
```

Here we see an example of both the assignment and access of an array element. With the first command, element 1 of array is assigned the value “foo”. The second command displays the stored value of element 1. The use of braces in the second command is required to prevent the shell from attempting pathname expansion on the name of the array element.

An array can also be created with the command:

```

```

Using the option, this example of creates the array .

Assigning Values To An Array

Values may be assigned in one of two ways. Single values may be assigned using the following syntax:

L

where is the name of the array and ! is an integer (or arithmetic expression) greater than or equal to zero. Note that the first element of an array is subscript zero, not one. is a string or integer assigned to the array element.

Multiple values may be assigned using the following syntax:

LM <<<O

where is the name of the array and ''' are values assigned sequentially to elements of the array, starting with element zero. For example, if we wanted to assign abbreviated days of the week to the array , , we could do this:

```
                    ! $4F  O  +  W  +  7  F  6
```

It is also possible to assign values to a specific element by specifying a subscript for each value:

```
                    ! $4(*)$F  ( )$O  ( )$+  ( )$W  (A)$+
```

```
(")$7   (N)$F  6
```

Accessing Array Elements

So what are arrays good for? Just as many data-management tasks can be performed with a spreadsheet program, many programming tasks can be performed with arrays.

Let’s consider a simple data-gathering and presentation example. We will construct a script that examines the modification times of the files in a specified directory. From this data, our script will output a table showing at what hour of the day the files were last modified. Such a script could be used to determine when a system is most active. This script, called , produces this result:

```
R          R

$$   $     "*    ""
$"   "     "/   0
$*   $     "&   "
$/   $     "#   0
$&   "     "+   +
$#   "     "0   #
$+   +     "!   &
$0   /     "%   &
$!   "     *$   "
$%   "&     *"   $
"$   *     **   $
""   #     */   $

)          L !$
```

We execute the program, specifying the current directory as the target. It produces a table showing, for each hour of the day (0-23), how many files were last modified. The code to produce this is as follows:

```
ZF5   5

Z          6              ,

    MO \
          _       \$ZZD5]        ,_ CY*
]
```

```
Z                          ,
    F    "   T

         "


Z @       B     ,
       \$<<*/]T          L$T

Z
        M        4, _ "_5D P        "* "/OT
       L \ 5Z$]
      MM77        OO
      MM77     OO


Z 9  6  ,
      _R   ^      ^ R   ^      _
      _     ^     ^     ^       _
       \$<<""]T
       L MM  7 "*OO
      6      _4$* ^ 4 ^ 4$* ^ 4 ^ _     \       ]     \       ]

6     _^ )          L 4 ^ _
```

The script consists of one function ( ) and a main body with four sections. In the first section, we check that there is a command line argument and that it is a directory. If it is not, we display the usage message and exit.

The second section initializes the array . It does this by assigning each element a value of zero. There is no special requirement to prepare arrays prior to use, but our script needs to ensure that no element is empty. Note the interesting way the loop is constructed. By employing brace expansion (\$<<*/]), we are able to easily generate a sequence of words for the command.

The next section gathers the data by running the program on each file in the directory. We use to extract the two-digit hour from the result. Inside the loop, we need to remove leading zeros from the hour field, since the shell will try (and ultimately fail) to interpret values “00” through “09” as octal numbers (see Table 34-2). Next, we increment the value of the array element corresponding with the hour of the day. Finally, we increment a counter ( ) to track the total number of files in the directory.

The last section of the script displays the contents of the array. We first output a couple of header lines and then enter a loop that produces four columns of output. Lastly, we output the final tally of files.

Array Operations

There are many common array operations. Such things as deleting arrays, determining their size, sorting, etc. have many applications in scripting.

Outputting The Entire Contents Of An Array

The subscripts D and can be used to access every element in an array. As with positional parameters, the notation is the more useful of the two. Here is a demonstration:

```
$4J    J J    J J    J6
 39     ( )>#       3 #



 39     (V)>#       3 #



 J39     ( )>J#       3 #

 J39     (V)>J#       3 #
```

We create the array and assign it three two-word strings. We then execute four loops to see the affect of word-splitting on the array contents. The behavior of notations \ D] and \ ] is identical until they are quoted. The D notation results in a single word containing the array’s contents, while the notation results in three two-word strings, which matches the array's “real” contents.

Determining The Number Of Array Elements

Using parameter expansion, we can determine the number of elements in an array in much the same way as finding the length of a string. Here is an example:

```
                        ( **)$
                           39U (V)>  U            !
"
                           39U ( **)>  U               **
/
```

We create array and assign the string “foo” to element 100. Next, we use parameter expansion to examine the length of the array, using the notation. Finally, we look at the length of element 100 which contains the string “foo”. It is interesting to note that while we assigned our string to element 100, bash only reports one element in the array. This differs from the behavior of some other languages in which the unused elements of the array (elements 0-99) would be initialized with empty values and counted. In , array elements only exist if they have been assigned a value regardless of their subscript.

Finding The Subscripts Used By An Array

As allows arrays to contain “gaps” in the assignment of subscripts, it is sometimes useful to determine which elements actually exist. This can be done with a parameter expansion using the following forms:

\F D]

\F]

where is the name of an array variable. Like the other expansions that use D and , the form enclosed in quotes is the most useful, as it expands into separate words:

```
                        $4( )$  (A)$  (N)$ 6
                            J39   (V)>J#        3 #



                            J39L   (V)>J#        3 #
*
&
+
```

Adding Elements To The End Of An Array

Knowing the number of elements in an array is no help if we need to append values to the end of an array, since the values returned by the D and notations do not tell us the maximum array index in use. Fortunately, the shell provides us with a solution. By using the 7L assignment operator, we can automatically append values to the end of an array. Here,

we assign three values to the array , and then append three more.

```
                    $4    6
                     39   (V)>

                    5$4    6
                     39   (V)>
```

Sorting An Array

Just as with spreadsheets, it is often necessary to sort the values in a column of data. The shell has no direct way of doing this, but it's not hard to do with a little coding:

```
ZF5  5

Z    ,     (        ,

 LM        O

   _S        ,  \   ]_
 N    LM M      _ \   ]_T       T     P    OO
   _(       ,    \ N      ]_
```

When executed, the script produces this:

```
                        !
S         ,
(        ,
```

The script operates by copying the contents of the original array ( ) into a second array (N) with a tricky piece of command substitution. This basic technique can be used to perform many kinds of operations on the array by changing the design of the pipeline.

Deleting An Array

To delete an array, use the command:

```
                    $4        6
                     39   (V)>


                     39   (V)>
```

may also be used to delete single array elements:

```
                    $4        6
                     39   (V)>

                      %   ( )%
                     39   (V)>
```

In this example, we delete the third element of the array, subscript 2. Remember, arrays start with subscript zero, not one! Notice also that the array element must be quoted to prevent the shell from performing pathname expansion.

Interestingly, the assignment of an empty value to an array does not empty its contents:

```
                    $4        6
                    $
                     39   (V)>
```

Any reference to an array variable without a subscript refers to element zero of the array:

```
                    $4        6
                     39   (V)>

                    $:
                     39   (V)>
3
```

Associative Arrays

Recent versions of now support . Associative arrays use strings

rather than integers as array indexes. This capability allow interesting new approaches to managing data. For example, we can create an array called “colors” and use color names as indexes:

```
 3
_ _L_Z $$$$_
_  _L_Z$$ $$_
_  _L_Z$$$$ _
```

Unlike integer indexed arrays, which are created by merely referencing them, associative arrays must be created with the command using the new 3 option. Associative array elements are accessed in much the same way as integer indexed arrays:

```
\   _  _]
```

In the next chapter, we will look at a script that makes good use of associative arrays to produce an interesting report.

Summing Up

If we search the man page for the word “array,” we find many instances of where makes use of array variables. Most of these are rather obscure, but they may provide occasional utility in some special circumstances. In fact, the entire topic of arrays is rather under-utilized in shell programming owing largely to the fact that the traditional Unix shell programs (such as) lacked any support for arrays. This lack of popularity is unfortunate because arrays are widely used in other programming languages and provide a powerful tool for solving many kinds of programming problems.

Arrays and loops have a natural affinity and are often used together. The

MM T T OO

form of loop is particularly well-suited to calculating array subscripts.

Further Reading

A couple of Wikipedia articles about the data structures found in this chapter:
http://en.wikipedia.org/wiki/Scalar_(computing)

http://en.wikipedia.org/wiki/Associative_array

, !

In this, the final chapter of our journey, we will look at some odds and ends. While we have certainly covered a lot of ground in the previous chapters, there are many features that we have not covered. Most are fairly obscure, and useful mainly to those integrating into a Linux distribution. However, there are a few that, while not in common use, are helpful for certain programming problems. We will cover them here.

Group Commands And Subshells

allows commands to be grouped together. This can be done in one of two ways; either with a or with a ! . Here are examples of the syntax of each:

Group command:

\ "T *T /T <<<]

Subshell:

M "T *T /T<<< O

The two forms differ in that a group command surrounds its commands with braces and a subshell uses parentheses. It is important to note that, due to the way implements group commands, the braces must be separated from the commands by a space and the last command must be terminated with either a semicolon or a newline prior to the closing brace.

So what are group commands and subshells good for? While they have an important difference (which we will get to in a moment), they are both used to manage redirection. Let's consider a script segment that performs redirections on multiple commands:

```
 C  6 <
_=      < _CC  6 <
 <  CC  6 <
```

This is pretty straightforward. Three commands with their output redirected to a file named 6 < . Using a group command, we could code this as follows:

```
\   T   _=      < _T    < T]C  6 <
```

Using a subshell is similar:

```
M   T   _=      < _T    < OC  6 <
```

Using this technique we have saved ourselves some typing, but where a group command or subshell really shines is with pipelines. When constructing a pipeline of commands, it is often useful to combine the results of several commands into a single stream. Group commands and subshells make this easy:

```
\   T   _=      < _T    < T]P 6
```

Here we have combined the output of our three commands and piped them into the input of 6 to produce a printed report.

In the script that follows, we will use groups commands and look at several programming techniques that can be employed in conjunction with associative arrays. This script, called , * , when given the name of a directory, prints a listing of the files in the directory along with the names of the file's owner and group owner. At the end of the listing, the script prints a tally of the number of files belonging to each owner and group. Here we see the results (condensed for brevity) when the script is given the directory 5 5 :

```
                    !
5  5  5* /*<+
5  5  5* /
5  5  5*6
5  5  5  8
5  5  5
5  5  5 6N   ,
5  5  5 6N
5  5  5   6  6   ,
<
<
<
5  5  5B6  6
5  5  5B6
5  5  5B6
5  5  5B6
```

```
5  5  5B 6 6
5  5  5B
5  5  5B

   8
        "    M O
       "/%&    M O

     6  8
        "    M O
        "    M O
6       "    M O
        &    M O
        "    M O
       "/!$    M O
  8      *    M O
        "    M O
,       *    M O
 6      *    M O
```

Here is a listing (with line numbers) of the script:

```
  "   ZF5   5
  *
  /   Z    , *  2      ,       ,      8
  &
  #         3        N    6     N 8      6   8
  +
  0       F    _ "_   T
  !         _2        , *    _ CY*
  %         "
  "$
  ""
  "*          _ "_5DT
  "/       8   L M       42 _  _O
  "&          6L M       4? _  _O
  "#           _ _ L_ _
  "+          N 8    _  _ L  8
  "0          N    6 _  _ L     6
  "!       MM77 8      8    OO
  "%       MM77    6      6 OO
  *$
  *"
  **    Z =
  */   \        _ \       ]_T
  *&       6      _4 &$  4 "$  4 "$ ^ _ ^
  *#          _  _  \    N 8    _  _ ] \    N    6 _  _ ]
  *+        ] P
```

```
*0
*!
*%   Z =    8
/$       _     8    _
/"   \       _ \F 8     ]_T
/*      6     _4 "$   4#     M O^ _ _  _  \ 8     _  _ ]
//       ] P
/&
/#
/+   Z =      6
/0       _       6  8    _
/!   \       _ \F   6   ]_T
/%      6     _4 "$   4#     M O^ _ _  _  \   6  _  _ ]
&$       ] P
```

Let's take a look at the mechanics of this script:

<6 Associative arrays must be created with the command using the 3 option. In this script we create five arrays as follows:

contains the names of the files in the directory, indexed by filename

N 6 contains the group owner of each file, indexed by filename

N 8 contains the owner of each file, indexed by file name

6 contains the number of files belonging to the indexed group

8 contains the number of files belonging to the indexed owner

= >6 Checks to see that a valid directory name was passed as a positional parameter. If not, a usage message is displayed and the script exits with an exit status of 1.

>6 Loop through the files in the directory. Using the command, lines 13 and 14 extract the names of the file owner and group owner and assign the values to their respective arrays (lines 16, 17) using the name of the file as the array index. Likewise the file name itself is assigned to the array (line 15).

? @6 The total number of files belonging to the file owner and group owner are incremented by one.

=6 The list of files is output. This is done using the "${array[@]}" parameter expansion which expands into the entire list of array elements with each element treated as a separate word. This allows for the possibility that a file name may contain embedded spaces. Also note that the entire loop is enclosed in braces thus forming a group command. This permits the entire output of the loop to be piped into the command. This is necessary because the expansion of the array elements is not sorted.

@ &>6 These two loops are similar to the file list loop except that they use the "${!

array[@]}" expansion which expands into the list of array indexes rather than the list of array elements.

Process Substitution

While they look similar and can both be used to combine streams for redirection, there is an important difference between group commands and subshells. Whereas a group command executes all of its commands in the current shell, a subshell (as the name suggests) executes its commands in a child copy of the current shell. This means that the environment is copied and given to a new instance of the shell. When the subshell exits, the copy of the environment is lost, so any changes made to the subshell's environment (including variable assignment) is lost as well. Therefore, in most cases, unless a script requires a subshell, group commands are preferable to subshells. Group commands are both faster and require less memory.

We saw an example of the subshell environment problem in Chapter 28, when we discovered that a command in a pipeline does not work as we might intuitively expect. To recap, if we construct a pipeline like this:

```
_ _ P
 Q':=U
```

The content of the Q':=U variable is always empty because the command is executed in a subshell, and its copy of Q':=U is destroyed when the subshell terminates.

Because commands in pipelines are always executed in subshells, any command that assigns variables will encounter this issue. Fortunately, the shell provides an exotic form of expansion called ! that can be used to work around this problem.

Process substitution is expressed in two ways:

For processes that produce standard output:

XM O

or, for processes that intake standard input:

CM O

where is a list of commands.

To solve our problem with , we can employ process substitution like this:

```
X XM  _  _O
 Q':=U
```

Process substitution allows us to treat the output of a subshell as an ordinary file for purposes of redirection. In fact, since it is a form of expansion, we can examine its real value:

```
                    XM    _  _O
5  5  5+/
```

By using to view the result of the expansion, we see that the output of the subshell is being provided by a file named 5 5 5+/ .

Process substitution is often used with loops containing . Here is an example of a loop that processes the contents of a directory listing created by a subshell:

```
ZF5  5

Z 6          6

8            8     6  B            T
       XX  'S

          ( B        B
          S8         8
          ?  6         6
          -
          =
          3

    'S
   X XM     P       7*O
```

The loop executes for each line of a directory listing. The listing itself is produced on the final line of the script. This line redirects the output of the process substitution into the standard input of the loop. The command is included in the process substitution pipeline to eliminate the first line of the listing, which is not needed.

When executed, the script produces output like this:

```
                         0       *
             <
( B       "&#&$
S8
?  6
-        *$$% $& $* "" "*
```

```
=       "
3        8


( B      &$%+
S8
?  6
-       *$$% $0 "$ $0 /"
=       *
3        8

             <
( B      /%&*"/
S8
?  6
```

Traps

In Chapter 10, we saw how programs can respond to signals. We can add this capability to our scripts, too. While the scripts we have written so far have not needed this capability (because they have very short execution times, and do not create temporary files), larger and more complicated scripts may benefit from having a signal handling routine.

When we design a large, complicated script, it is important to consider what happens if the user logs off or shuts down the computer while the script is running. When such an event occurs, a signal will be sent to all affected processes. In turn, the programs representing those processes can perform actions to ensure a proper and orderly termination of the program. Let’s say, for example, that we wrote a script that created a temporary file during its execution. In the course of good design, we would have the script delete the file when the script finishes its work. It would also be smart to have the script delete the file if a signal is received indicating that the program was going to be terminated prematurely.

provides a mechanism for this purpose known as a . Traps are implemented with the appropriately named builtin command, 6 . 6 uses the following syntax:

6 <<<

where is a string which will be read and treated as a command and is the specification of a signal that will trigger the execution of the interpreted command.

Here is a simple example:

```
ZF5  5

Z  6      6
```

```
 6 _    J@          ,  <J_ (@?@K) (@?)'Q-

     \"<<#]T
       _@           #_
      6 #
```

This script defines a trap that will execute an command each time either the SIGINT or SIGTERM signal is received while the script is running. Execution of the program looks like this when the user attempts to stop the script by pressing :

```
@      "   #
@      *   #
@        , <
@      /   #
@        , <
@      &   #
@      #   #
```

As we can see, each time the user attempts to interrupt the program, the message is printed instead.

Constructing a string to form a useful sequence of commands can be awkward, so it is common practice to specify a shell function as the command. In this example, a separate shell function is specified for each signal to be handled:

```
ZF5  5

Z   6   *          6

  N  N     N(@?@K) MO \
       _(  6      6  <_ *CY"
       $
]

  N  N     N(@?)'Q- MO \
       _(  6        <_ *CY"
       $
]

 6    N  N     N(@?@K) (@?@K)
 6    N  N     N(@?)'Q- (@?)'Q-
```

```
\"<<#]T
  _@          #_
 6 #
```

This script features two 6 commands, one for each signal. Each trap, in turn, specifies a shell function to be executed when the particular signal is received. Note the inclusion of an command in each of the signal-handling functions. Without an , the script would continue after completing the function.

When the user presses during the execution of this script, the results look like this:

```
@     "  #
@     *  #
( 6     6 <
```

Temporary Files

One reason signal handlers are included in scripts is to remove temporary files that the script may create to hold intermediate results during execution. There is something of an art to naming temporary files. Traditionally, programs on Unix-like systems create their temporary files in the 5 6 directory, a shared directory intended for such files. However, since the directory is shared, this poses certain security concerns, particularly for programs running with superuser privileges. Aside from the obvious step of setting proper permissions for files exposed to all users of the system, it is important to give temporary files non-predictable filenames. This avoids an exploit known as a . One way to create a non-predictable (but still descriptive) name is to do something like this:

```
6   L5  65 M       $O<  < Q3K9S-
```

This will create a filename consisting of the program's name, followed by its process ID (PID), followed by a random integer. Note, however, that the Q3K 9S- shell variable only returns a value in the range of 1-32767, which is not a very large range in computer terms, so a single instance of the variable is not sufficient to overcome a determined attacker.

A better way is to use the 6 program (not to be confused with the 6 standard library function) to both name and create the temporary file. The 6 program accepts a template as an argument that is used to build the filename. The template should include a series of "X" characters, which are replaced by a corresponding number of random letters and numbers. The longer the series of "X" characters, the longer the series of random characters. Here is an example:

```
6  L M   6 5  65    <  <VVVVVVVVVO
```

This creates a temporary file and assigns its name to the variable 6 . The "X" characters in the template are replaced with random letters and numbers so that the final filename (which, in this example, also includes the expanded value of the special parameter to obtain the PID) might be something like:

```
5  65    <+#%/<2SH  -++#&
```

For scripts that are executed by regular users, it may be wise to avoid the use of the 5 6 directory and create a directory for temporary files within the user's home directory, with a line of code such as this:

```
RS-'5  6   PP      RS-'5  6
```

Asynchronous Execution

It is sometimes desirable to perform more than one task at the same time. We have seen how all modern operating systems are at least multitasking if not multiuser as well. Scripts can be constructed to behave in a multitasking fashion.

Usually this involves launching a script that, in turn, launches one or more child scripts that perform an additional task while the parent script continues to run. However, when a series of scripts runs this way, there can be problems keeping the parent and child coordinated. That is, what if the parent or child is dependent on the other, and one script must wait for the other to finish its task before finishing its own?

has a builtin command to help manage such as this. The 8 command causes a parent script to pause until a specified process (i.e., the child script) finishes.

8

We will demonstrate the 8 command first. To do this, we will need two scripts, a parent script:

```
ZF5   5

Z   ,   6     3 ,                M6    O

    _:          <<<_

    _:                  6 <<<_
  ,      Y
6  L F
    _:         M:@9L  6  O       <_

    _:            <<<_
   6 *

    _:     6       8               <<<_
8    6

    _:                <        <<<_
    _:     6         < '     <_
```

and a child script:

```
ZF5   5

Z   ,       3 ,                M    O

    _                <<<_
   6 #
    _              < '     <_
```

In this example, we see that the child script is very simple. The real action is being performed by the parent. In the parent script, the child script is launched and put into the background. The process ID of the child script is recorded by assigning the 6 variable with the value of the F shell parameter, which will always contain the process ID of the last job put into the background.

The parent script continues and then executes a 8 command with the PID of the child process. This causes the parent script to pause until the child script exits, at which point the parent script concludes.

When executed, the parent and child scripts produce the following output:

```
                        !
:          <<<
```

```
:                 6 <<<
:         M:@9L +0&"O       <
:            <<<
                <<<
:    6       8                <<<
             < '    <
:                <        <<<
:    6         < '    <
```

Named Pipes

In most Unix-like systems, it is possible to create a special type of file called a . Named pipes are used to create a connection between two processes and can be used just like other types of files. They are not that popular, but they're good to know about.

There is a common programming architecture called , which can make use of a communication method such as named pipes, as well as other kinds of such as network connections.

The most widely used type of client-server system is, of course, a web browser communicating with a web server. The web browser acts as the client, making requests to the server and the server responds to the browser with web pages.

Named pipes behave like files, but actually form first-in first-out (FIFO) buffers. As with ordinary (unnamed) pipes, data goes in one end and emerges out the other. With named pipes, it is possible to set up something like this:

C

and

X

and it will behave as if:

P

Setting Up A Named Pipe

First, we must create a named pipe. This is done using the command:

```
6 8      "          $ *$$% $0 "0 $+ &" 6 6 "
```

Here we use to create a named pipe called 6 6 " . Using , we examine the file and see that the first letter in the attributes field is “p”, indicating that it is a named pipe.

Using Named Pipes

To demonstrate how the named pipe works, we will need two terminal windows (or alternately, two virtual consoles). In the first terminal, we enter a simple command and redirect its output to the named pipe:

```
                    &
```

After we press the ' key, the command will appear to hang. This is because there is nothing receiving data from the other end of the pipe yet. When this occurs, it is said that the pipe is ! . This condition will clear once we attach a process to the other end and it begins to read input from the pipe. Using the second terminal window, we enter this command:

```
                    /
```

and the directory listing produced from the first terminal window appears in the second terminal as the output from the command. The command in the first terminal successfully completes once it is no longer blocked.

Summing Up

Well, we have completed our journey. The only thing left to do now is practice, practice, practice. Even though we covered a lot of ground in our trek, we barely scratched the surface as far as the command line goes. There are still thousands of command line programs left to be discovered and enjoyed. Start digging around in 5 5 and you’ll see!

Further Reading

The “Compound Commands” section of the bash man page contains a full description of group command and subshell notations.

The EXPANSION section of the bash man page contains a subsection covering process substitution.

* 9 # ? also has a discussion of process substitution:
http://tldp.org/LDP/abs/html/process-sub.html

D has two good articles on named pipes. The first, from September 1997:
http://www.linuxjournal.com/article/2156

and the second, from March 2009:
http://www.linuxjournal.com/content/using-named-pipes-fifos-bash

#

a2ps command.....337
absolute pathnames.....9
alias command.....50, 128
aliases.....42, 50, 126
American National Standards Institute (see ANSI) 162
American Standard Code for Information Interchange (see ASCII).....18
anchors.....251
anonymous FTP servers.....203
ANSI.....162
ANSI escape codes.....162, 166
ANSI.SYS.....162
Apache web server.....119
apropos command.....47
apt-cache command.....171
apt-get command.....170p.
aptitude command.....170
Arch.....168
archiving.....234
arithmetic expansion.....70, 75, 369, 459, 467
arithmetic expressions.....70, 456, 467, 470
arithmetic operators.....70, 468
arithmetic truth tests.....393, 467
arrays.....
  append values to the end.....486
  assigning values.....482
  associative.....488, 491
  creating.....481
  deleting.....487
  determine number of elements.....485
  finding used subscripts.....486
  index.....481
  multidimensional.....481
  reading variables into.....403
  sorting.....487
  subscript.....481
  two-dimensional.....481
ASCII.....78, 82, 225, 255, 267, 337
  bell character.....159
  carriage return.....271
  collation order.....255, 257, 389
  control codes.....78, 255, 331
  groff output driver.....324
  linefeed character.....271
  null character.....225
  printable characters.....255
  text.....18
aspell command.....303
assembler.....344
assembly language.....344
assignment operators.....470
associative arrays.....488, 491
asynchronous execution.....499
audio CDs.....182, 194
AWK programming language.....303, 476

1

back references.....267, 298, 300
backslash escape sequences.....78
backslash-escaped special characters.....158
backups, incremental.....238
basename command.....443
bash.....2, 126
  man page.....48
basic regular expressions 258, 266p., 296, 300, 310
bc command.....476
Berkeley Software Distribution.....335
bg command.....117
binary.....94, 98, 344, 468
bit mask.....97
bit operators.....472
Bourne, Steve.....2, 6
brace expansion.....71, 75, 454

branching....383
break command....415, 448
broken links....40
BSD style....112
buffering....184
bugs....426p.
build environment....349
bzip2 command....233

!
C programming language....344, 456, 471, 474
C++....344
cal command....4
cancel command....342
carriage return. .18, 78, 159, 255p., 270, 302p., 334
case compound command....432
case conversion....465
cat command....58, 270
cd command....9, 11
CD-ROMs....181p., 193
cdrecord command....194
cdrtools....194
character classes…26p., 252, 254p., 257, 261, 294, 303
character ranges....27, 253p., 303
chgrp command....104
child process....109
chmod command....93, 106, 358
chown command....103, 106
Chrome....363
chronological sorting....277
cleartext....203, 206
client-server architecture....501
COBOL programming language....344
collation order....128, 255, 257, 293, 389
  ASCII....257, 389
  dictionary....255
  traditional....257
comm command....288
command history....3, 84
command line....
  arguments....439
  editing....3, 80
  expansion....67
  history....3, 85
  interfaces....xvii, 28
command options....14
command substitution....73, 75, 454
commands....
  arguments....14, 439
  determining type....43
  documentation....44
  executable program files....42, 344
  executing as another user....100
  long options....14
  options....14
comments....130, 136, 302, 357, 427
Common Unix Printing System....333, 342
comparison operators....473
compiler....344
compiling....343
completions....82
compound commands....
  case....432
  for....453
  if....383
  until....416
  while....413
  (( ))....393, 409, 467
  [[ ]]....391, 409
compression algorithms....231
conditional expressions....399, 423
configuration files....18, 21, 126
configure command....349
constants....368
continue command....415
control characters....159, 270
control codes....78, 255
control operators....
  &&....397, 409
  ||....397
controlling terminal....110
COPYING....347
copying and pasting....
  in vim....147
  on the command line....81
  with X Window System....3
coreutils package....45, 48p., 283, 307
counting words in a file....62
cp command....28, 35, 134, 210
CPU....109p., 343
cron job....215
crossword puzzles....251
csplit command....308
CUPS....333, 342
current working directory....8
cursor movement....80
cut command....281, 464

/

daemon programs.....109, 119
data compression.....230
data redundancy.....230
data validation.....392
date command.....4
date formats.....277
dd command.....192
Debian.....168
Debian Style (.deb).....169
debugging.....379, 427
declare command.....466
defensive programming.....423, 427
delimiters.....76, 275, 278
dependencies.....170, 352
design.....425, 427
device drivers.....176, 344
device names.....184
device nodes.....20
df command.....4, 381
diction.....345
dictionary collation order.....255
diff command.....289
Digital Restrictions Management (DRM).....170
directories.....
  archiving.....234
  changing.....9
  copying.....28
  creating.....28, 34
  current working.....8
  deleting.....31, 39
  hierarchical.....7
  home.....21, 91, 381
  listing.....13
  moving.....30, 36
  navigating.....7
  OLD_PWD variable.....128
  parent.....8
  PATH variable.....128
  PWD variable.....129
  removing.....31, 39
  renaming.....30, 36
  root.....7
  shared.....105
  sticky bit.....99
  synchronizing.....242
  transferring over a network.....242
  viewing contents.....8
disk partitions.....179
DISPLAY variable.....128
Dolphin.....27
dos2unix command.....271
double quotes.....75
dpkg command.....170
du command.....273, 381
Dynamic Host Configuration Protocol (DHCP) 202

$
echo command.....67, 127, 364
  -e option.....78
  -n option.....401
edge and corner cases.....426
EDITOR variable.....128
effective group ID.....99
effective user ID.....99, 110
elif statement.....390
email.....269
embedded systems.....344
empty variables.....460
encrypted tunnels.....210
encryption.....294
end of file.....59, 371
endless loop.....416
enscript command.....340
environment.....101, 126, 407
  aliases.....126
  establishing.....129
  examining.....126
  login shell.....129
  shell functions.....126
  shell variables.....126
  startup files.....129
  subshells.....494
  variables.....126
eqn command.....322
executable files.....350
executable program files.....42, 344
executable programs.....
  determining location.....43
  PATH variable.....128
exit command.....5, 388, 410
exit status.....384, 388
expand command.....283
expansions.....67
  arithmetic.....70, 75, 369, 459, 467
  brace.....71, 75, 454
  command substitution.....73, 75, 454
  delimiters.....76
  errors resulting from.....421
  history.....85, 87
  parameter.....72, 75, 367, 373, 459
  pathname.....68, 75, 454

tilde....69, 75
word-splitting....75
expressions....
arithmetic....70, 456, 467, 470, 482
conditional....399, 423
ext4....190
extended regular expressions....258
Extensible Markup Language....269

9

false command....385
fdformat command....192
fdisk command....187
fg command....117
FIFO....501
file command....17
file descriptor....56
file system corruption....184
File Transfer Protocol (FTP)....203
filenames....225
case sensitive....11
embedded spaces in....12, 264
extensions....12
hidden....11
files....
access....90
archiving....234, 240
attributes....91
block special....92
block special device....216
changing file mode....93
changing owner and group owner....103
character special....92
character special device....216
compression....230
configuration....18, 126, 268
copying....28, 35
copying over a network....203
creating empty....55
deb....168
deleting....31, 39, 222
determining contents....17
device nodes....20
execution access....91
expressions....386
finding....213
hidden....11
iso image....193, 195
listing....8, 13
mode....92
moving....30, 36
owner....93
permissions....90
read access....91
regular....216
removing....31, 39
renaming....30, 36
rpm....168
shared library....21
startup....129
sticky bit....99
symbolic links....216
synchronizing....242
temporary....498
text....17
transferring over a network....203, 239, 242
truncating....55
type....91
viewing contents....17
write access....91
filters....61
find command....215, 238
findutils package....229
Firefox....363
firewalls....199
first-in first-out....501
floppy disks....178, 185, 192
flow control....
branching....383
case compound command....432
elif statement....390
endless loop....416
for compound command....453
for loop....453
function statement....376
if compound command....383
looping....412
menu-driven....409
multiple-choice decisions....432
reading files with while and until loops....417
terminating a loop....415
traps....496
until loop....416
while loop....413
fmt command....313
focus policy....4
fold command....313
for compound command....453
for loop....453
Fortran programming language....344, 456
free command....5, 183

Free Software Foundation.....xix, xxi
fsck command.....191
ftp command.....203, 211, 345, 372
FTP servers.....203, 372
FUNCNAME variable.....444
function statement.....376

A

gcc.....345
gedit command.....115, 133
genisoimage command.....194
Gentoo.....168
getopts command.....452
Ghostscript.....333
gid.....90
global variables.....378
globbing.....26
GNOME.....2, 27, 41, 97, 133, 211
gnome-terminal.....2
GNU binutils package.....455
GNU C Compiler.....345
GNU coreutils package.....45, 48p., 283, 307
GNU findutils package.....229
GNU Project.....14, xix, xxi, 229, 307, 345, 347
  info command.....48
GNU/Linux.....xix, xxi
graphical user interfaces.....xvii
grep command.....62, 247, 406
groff.....322
group commands.....490
groups.....90
  effective group ID.....99
  gid.....90
  primary group ID.....90
  setgid.....99
GUI.....3, xvii, 27, 41, 80, 96, 129
gunzip command.....231
gzip command.....50, 231

halt command.....122
hard disks.....178
hard links.....24, 33, 37
  creating.....37
  listing.....38
head command.....63
header files.....348
hello world program.....357
help command.....44
here documents.....371
here strings.....407
hexadecimal.....94, 468
hidden files.....11, 69
hierarchical directory structure.....7
high-level programming languages.....344
history.....
  expansion.....85, 87
  searching.....85
history command.....85
home directories.....21
  root account.....22
  /etc/passwd.....91
home directory.....8, 11, 69, 101, 128
HOME variable.....128
hostname.....159
HTML.....269, 303, 323, 363, 373, 375
Hypertext Markup Language.....269

)

I/O redirection (see redirection).....53
ICMP ECHO_REQUEST.....199
id command.....90
IDE.....185
if compound command.....131, 421, 432
ifconfig command.....201
IFS variable.....405
incremental backups.....238
info files.....49
init.....109
init scripts.....109
inodes.....38
INSTALL.....347
installation wizard.....169
integers.....
  arithmetic.....70, 476
  division.....71, 469
  expressions.....390
interactivity.....400
Internal Field Separator.....405
interpreted languages.....344
interpreted programs.....345
interpreter.....344
ip command.....201
iso images.....193, 195
iso9660.....182, 195

B

job control.....116

job numbers....116
jobspec....117
join command....285
Joliet extensions....194
Joy, Bill....139

4

kate command....133
KDE....2, 27, 41, 97, 133, 211
kedit command....133
kernel...xvi, xixp., 46, 109, 119, 176, 185, 291, 353
key fields....275
kill command....118
killall command....121
killing text....81
Knuth, Donald....322
Konqueror....27, 97, 211
konsole....2
kwrite command....115, 133

LANG variable....128, 255, 257
less command....17, 60, 242, 265
lftp command....205
libraries....344
LibreOffice Writer....18, xxi
line continuation character....361
line editors....139
line-continuation character....302
linker....344
linking....344
links....
  broken....40
  creating....33
  hard....24, 33
  symbolic....23, 34
Linux community....168
Linux distributions....168
  Arch....168
  CentOS....169
  Debian....168p., 343
  Fedora....xix, 90, 169
  Gentoo....168
  Linux Mint....169
  OpenSUSE....xix, 169
  packaging systems....168
  Raspbian....169
  Red Hat Enterprise Linux....169
  Slackware....168
  Ubuntu....xix, 168p.
Linux Filesystem Hierarchy Standard...19, 24, 360
Linux kernel.....xvi, xixp., 46, 109, 119, 176, 185, 291, 353
  device drivers....176
literal characters....249
live CDs....xix
ln command....33, 37
local variables....378
locale....255, 257, 293, 389
locale command....257
localhost....207
locate command....213, 265
logical errors....423
logical operations....394
logical operators....218
logical relationships....218, 222
login prompt....5, 204
login shell....91, 101, 129
long options....14
loopback interface....202
looping....412
loops....423, 469, 472, 489, 495
lossless compression....231
lossy compression....231
lowercase to uppercase conversion....466
lp command....336
lpq command....341
lpr command....335
lprm command....342
lpstat command....340
ls command....8, 13
  long format....16
  viewing file attributes....91
Lukyanov, Alexander....205
LVM (Logical Volume Manager)....178, 181

:

machine language....343
maintenance....360, 364, 366, 374
make command....350
Makefile....350
man command....45
man pages....45, 323
markup languages....269, 323
memory....
  assigned to each process....110
  displaying free....5
  Resident Set Size....112
  segmentation violation....120

usage....112
viewing usage....123
virtual....112
menu-driven programs....409
meta key....82
meta sequences....250
metacharacters....250
metadata....169, 171
mkdir command....28, 34
mkfifo command....501
mkfs command....190, 192
mkisofs command....194
mktemp command....499
mnemonics....344
modal editor....141
monospaced fonts....333
Moolenaar, Bram....139
more command....19
mount command....180, 195
mount points....21, 180, 182
mounting....179
MP3....105
multi-user systems....89
multiple-choice decisions....432
multitasking....89, 109, 499
mv command....30, 36

2

named pipes....501
nano command....138
Nautilus....27, 97, 211
netstat command....202
networking....198
anonymous FTP servers....203
default route....203
Dynamic Host Configuration Protocol (DHCP)....202
encrypted tunnels....210
examine network settings and statistics....202
File Transfer Protocol (FTP)....203
firewalls....199
FTP servers....203
Local Area Network....203
loopback interface....202
man in the middle attacks....206
routers....201
secure communication with remote hosts....206
testing if a host is alive....199
tracing the route to a host....200
transferring files....242
transporting files....203
Virtual Private Network....210
newline character....159
newlines....76
NEWS....347
nl command....309
nroff command....322
null character....225
number bases....468

;

octal....94, 468, 484
Ogg Vorbis....105
OLD_PWD variable....128
OpenOffice.org Writer....xxp.
OpenSSH....206
operators....
arithmetic....70, 468
assignment....470
binary....422
comparison....473
ternary....474
owning files....90

package files....169
package maintainers....169
package management....168
deb....168
Debian Style (.deb)....169
finding packages....171
high-level tools....170
installing packages....171
low-level tools....170
package repositories....169
Red Hat Style (.rpm)....169
removing packages....172
RPM....168
updating packages....173
packaging systems....168
page description language....269, 324, 332
PAGER variable....128
pagers....19
parameter expansion....72, 75, 459
parent directory....8
parent process....109
passwd command....107
passwords....107
paste command....284

PATA....185
patch command....291
patches....289
PATH variable....128, 131, 358, 376
pathname expansion....68, 75, 454
pathnames....264
  absolute....9
  completion....82
  relative....9
PDF....325, 335
Perl programming language....42, 247, 303, 344, 476
permissions....356
PHP programming language....344
ping command....199
pipelines....60, 407, 494
  in command substitution....73
portability....349, 382, 396
portable....382
Portable Document Format....325, 335
Portable Operating System Interface....259
positional parameters....439, 460p., 463
POSIX....194, 255, 258p., 396
  character classes....26p., 254p., 257, 261, 294, 303
PostScript....269, 324, 332, 337, 342
poweroff command....122
pr command....317, 333
primary group ID....90
printable characters....255
printenv command....73, 126
printer buffers....183
printers....183, 185
  buffering output....183
  control codes....331
  daisy-wheel....331
  device names....185
  drivers....333
  graphical....332
  impact....331
  laser....332
printf command....318, 458
printing....
  determining system status....340
  history of....330
  Internet Printing Protocol....341
  monospaced fonts....331
  preparing text....333
  pretty....337
  print queues....340
  proportional fonts....332
  queue....341
  spooling....340
  terminate print jobs....342
  viewing jobs....341
process ID....110
process substitution....494
processes....109
  background....116
  child....109
  controlling....114
  foreground....116
  interrupting....115
  job control....116
  killing....118
  nice....111
  parent....109
  PID....110
  process ID....110
  SIGINT....497
  signals....118
  SIGTERM....497
  sleeping....111
  state....111
  stopping....117
  viewing....110, 112
  zombie....111
production use....425
programmable completion....84
ps command....110
PS1 variable....128, 158
PS2 variable....365
ps2pdf command....325
PS4 variable....429
pseudocode....383, 412
pstree command....123
PuTTY....212
pwd command....8
PWD variable....129
Python programming language....344

C

quoting....74
  double quotes....75
  escape character....77
  missing quote....420
  single quotes....77

RAID (Redundant Array of Independent Disks)
....178

raster image processor.....333
read command.....401, 411, 417, 425, 494
Readline.....80
README.....49, 347
reboot command.....122
redirection.....
  blocked pipe.....502
  group commands and subshells.....490
  here documents.....371
  here strings.....407
  standard error.....55
  standard input.....57, 372
  standard output.....54
redirection operators.....
  &>.....57
  &>>.....57
  <.....59
  <(list).....494
  <<.....371p.
  <<-.....372
  <<<.....407
  >.....54
  >(list).....494
  >>.....55
  |.....60
regular expressions.....62, 247, 299, 392, 406
  anchors.....251
  back references.....267, 298, 300
  basic.....258, 266p., 296, 300, 310
  extended.....258
relational databases.....285
relative pathnames.....9
release early, release often.....425
removing duplicate lines in a file.....61
REPLY variable.....401, 494
report generator.....363
repositories.....169
return command.....377, 388
reusable.....382
RIP.....333
rlogin command.....206
rm command.....31
Rock Ridge extensions.....194
roff.....322
ROT13 encoding.....294
RPM.....168
rpm command.....170
rsync command.....242
rsync remote-update protocol.....242
Ruby programming language.....344
scalar variables.....481
Schilling, Jorg.....194
scp command.....210
script command.....87
scripting languages.....42, 344
sdiff command.....308
searching a file for patterns.....62
searching history.....85
Secure Shell.....206
sed command.....295, 326, 464
set command.....126, 428
setgid.....99
setuid.....99, 387
Seward, Julian.....233
sftp command.....211
shared libraries.....21, 170
shebang.....357, 362
shell builtins.....42
shell functions.....42, 126, 376, 443
shell prompts 2, 9, 86, 101, 115, 128, 158, 207, 365
shell scripts.....356
SHELL variable.....128
shell variables.....126
shift command.....442, 447
shutdown command.....122
SIGINT.....497
signals.....496
single quotes.....77
Slackware.....168
sleep command.....414
soft link.....23
sort command.....61, 272
sort keys.....275
source code.....168p., 176, 269, 343
source command.....137, 359
source tree.....346
special parameters.....444, 461
split command.....308
SSH.....206
ssh command.....207, 239
ssh program.....89
Stallman, Richard.....xvi, xix, xxi, 133, 259, 345
standard error.....53p., 56
  disposing of.....57
  redirecting to a file.....55
standard input.....53, 372, 401
  redirecting.....57
standard output.....53
  appending to a file.....55
  disposing of.....57

redirecting standard error to....56
redirecting to a file....54
startup files....129
stat command....227
sticky bit....99
storage devices....178
audio CDs....182, 194
CD-ROMs....181p., 193
creating file systems....187
device names....184
disk partitions....179
FAT32....187
floppy disks....185, 192
formatting....187
LVM (Logical Volume Manager)....181
mount points....180, 182
partitions....187
reading and writing directly....192
repairing file systems....191
unmounting....183
USB flash drives....193
stream editor....295
strings....
expressions....389
extract a portion of....462
length of....462
perform search and replace upon....464
remove leading portion of....463
remove trailing portion of....463
${parameter:offset:length}....462
${parameter:offset}....462
strings command....455
stubs....379, 426
style....348
su command....100
subshells....407, 490
sudo command....100, 102
Sun Microsystems....139
superuser....2, 91, 101, 121
symbolic links....23, 34, 38
creating....38, 41
listing....39
symlink....23
syntax errors....419
syntax highlighting....356, 361

tables....285
tabular data....275, 321
tail command....63
tape archive....235
tar command....234
tarballs....346
targets....350
Task Manager....114
Tatham, Simon....212
tbl command....322, 326
tee command....64
Teletype....110
telnet command....206
TERM variable....129
terminal emulators....2
terminal sessions....
controlling terminal....110
effect of .bashrc....359
environment....101
exiting....5
login shell....101, 129
TERM variable....129
using named pipes....502
virtual....5
with remote systems....89
terminals....82, 88p., 162, 322, 331
ternary operator....474
test cases....426
test command....386, 391, 422
test coverage....427
testing....425, 427
TEX....322
text....17
adjusting line length....313
ASCII....18
carriage return....271
comparing....288
converting MS-DOS to Unix....293
counting words....62
cutting....281
deleting duplicate lines....279
deleting multiple blank lines....271
detecting differences....289
displaying common lines....288
displaying control characters....270
DOS format....271
EDITOR variable....128
editors....132, 268
expanding tabs....283
files....17
filtering....61
folding....313
formatting....309
formatting for typesetters....322

formatting tables....326
joining....285
linefeed character....271
lowercase to uppercase conversion....293
numbering lines....271, 309
paginating....317
pasting....284
preparing for printing....333
removing duplicate lines....61
rendering in PostScript....324
ROT13 encoded....294
searching for patterns....62
sorting....61, 272
spell checking....303
substituting....298
substituting tabs for spaces....283
tab-delimited....282
transliterating characters....293
Unix format....271
viewing with less....17, 60
text editors....132, 268, 293
emacs....133
for writing shell scripts....356
gedit....133, 356
interactive....293
kate....133, 356
kedit....133
kwrite....133
line....139
nano....133, 138
pico....133
stream....295
syntax highlighting....356, 361
vi....133
vim....133, 356, 361
visual....139
tilde expansion....69, 75
tload command....123
top command....112
top-down design....374
Torvalds, Linus....xvi, xxi
touch command....226p., 244, 352, 449
tr command....293
traceroute command....200
tracing....428
transliterating characters....293
traps....496
troff command....322
true command....385
TTY....110
type command....43
typesetters....322, 332
TZ variable....129

-

Ubuntu....90, 103, 168, 254, 359
umask command....97, 106
umount command....183
unalias command....51
unary operator expected....422
unary operators....468
unexpand command....283
unexpected token....421
uniq command....61, 279
Unix....xvii
Unix System V....335
unix2dos command....271
unset command....487
until compound command....416
until loop....416
unzip command....240
updatedb command....215
upstream providers....169
uptime....375
uptime command....381
USB flash drives....178, 193
Usenet....294
USER variable....127, 129
users....
accounts....90
changing identity....100
changing passwords....107
effective user ID....99, 110
home directory....91
identity....90
password....91
setting default permissions....97
setuid....99
superuser....91, 93, 99p., 108
/etc/passwd....91
/etc/shadow....91
UUID....180

D

validating input....407
variables....73, 366, 459
assigning values....369, 470
constants....368
declaring....366, 369
environment....126

global....378
local....378
names....368, 462
scalar....481
shell....126
vfat....190
vi command....138
vim command....267, 361
virtual consoles....5
Virtual Private Network....210
virtual terminals....5
visual editors....139
vmstat command....123

'

wait command....499
wc command....62
web pages....269
wget command....205
What You See Is What You Get....331
whatis command....47
which command....43, 73
while compound command....413
wildcards....26, 58, 67, 247, 254
wodim command....195
word-splitting....75p.
world....90
WYSIWYG....331

E

X Window System....3, 89, 210
xargs command....224
xload command....123
xlogo command....115
XML....269

F

yanking text....81
yum command....170p.

G

zgrep command....267
zip command....240
zless command....50

--help option....45
./configure....349
.bash_history....85
.bash_login....129
.bash_profile....129
.bashrc....130, 132, 359, 382, 444
.profile....129
.ssh/known_hosts....208

*

(( )) compound command....467, 473

H

[command....421
[[]] compound command....413

I

/....20
/bin....20
/boot....20
/boot/grub/grub.conf....20
/boot/vmlinuz....20
/dev....20
/dev/cdrom....185
/dev/dvd....185
/dev/floppy....185
/dev/null....57
/etc....21
/etc/bash.bashrc....130
/etc/crontab....21
/etc/fstab....21, 179, 191
/etc/group....91
/etc/passwd....21, 91, 278, 283, 406
/etc/profile....129, 131
/etc/shadow....91
/etc/sudoers....100
/lib....21
/lost+found....21
/media....21
/mnt....21
/opt....21
/proc....22
/root....22, 101
/sbin....22
/tmp....22, 499
/usr....22
/usr/bin....22
/usr/lib....22

/usr/local....22
/usr/local/bin....22, 353, 360
/usr/local/sbin....360
/usr/sbin....22
/usr/share....22
/usr/share/dict....251
/usr/share/doc....22, 49
/var....23
/var/log....23
/var/log/messages....23, 64, 185
/var/log/syslog....23, 64, 185

J

$!....500
$((expression))....467
${!array[@]}....486
${!array[*]}....486
${!prefix@}....462
${!prefix*}....462
${#parameter}....462
${parameter,,}....466
${parameter,}....466
${parameter:-word}....460
${parameter:?word}....461
${parameter:+word}....461
${parameter:=word}....460
${parameter//pattern/string}....464
${parameter/#pattern/string}....464
${parameter/%pattern/string}....464
${parameter/pattern/string}....464
${parameter##pattern}....463
${parameter#pattern}....463
${parameter%%pattern}....463
${parameter%pattern}....463
${parameter^}....467
${parameter^^}....466
$@....444, 452
$*....444p., 452
$#....440
$0....444